**Devang Patel Institute of Advance Technology and Research**
DEPSTAR (A Constitute Institute of CHARUSAT)

# Certificate

*This is to certify that*

*Mr./~~Mrs.~~* Probin Bhagchandani

*of* 6CE1 *Class,*

*ID. No.* 22DCE006 *has satisfactorily completed*

*his/~~her~~ term work in* CE364 - Cryptography and Network Security *for*

*the ending in* April 2024/2025

*Date :* 1/4/25

*Sign. of Faculty*

*Head of Department*

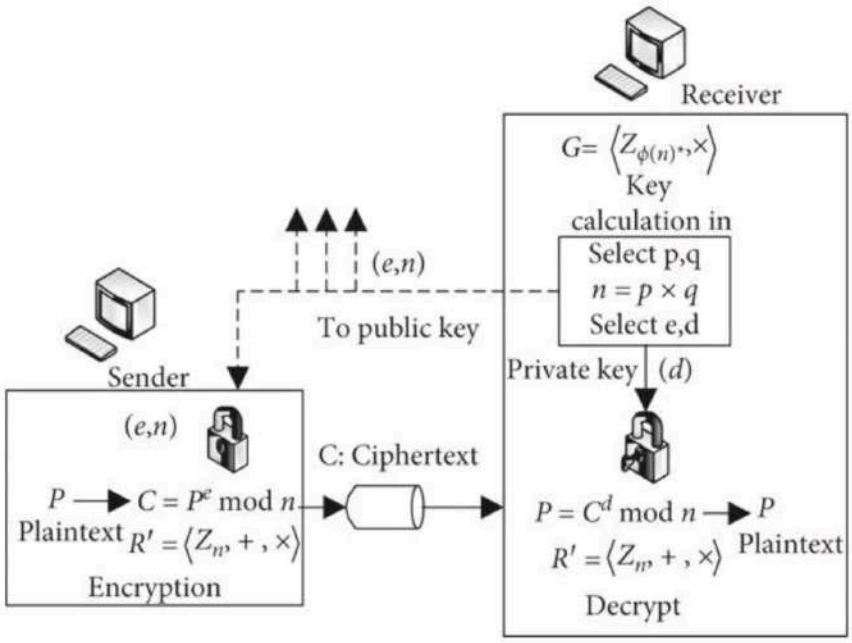
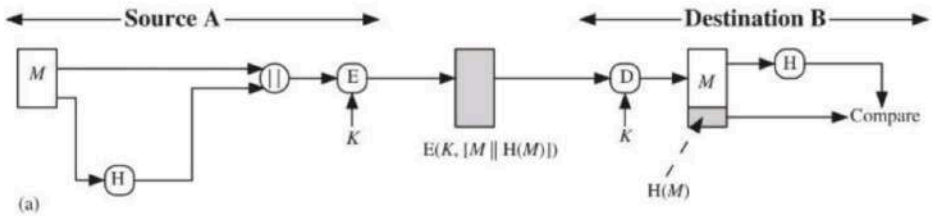
**CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY**
**FACULTY OF TECHNOLOGY AND ENGINEERING (FTE)**
**Chandubhai S. Patel Institute of Technology (CSPIT) &**
**Devang Patel Institute of Advance Technology and Research (DEPSTAR)**
**ACADEMIC YEAR: 2024-25**

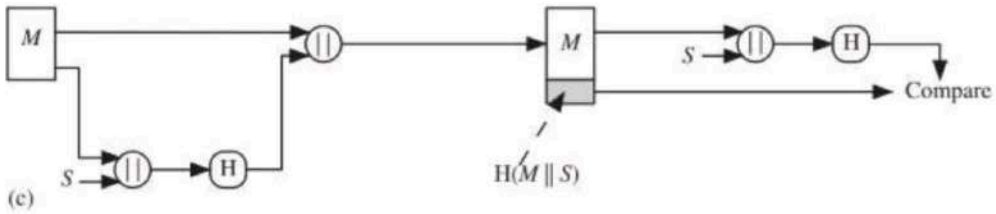
# Practical List Index

## Subject: CE364 Cryptography and Network Security (6th Sem)

| Exp. No. | Name of Experiment | Hours | CO |
|---|---|---|---|
| 1. | A security training institute is setting up a lab for ethical hacking workshops. The team must decide whether to use Kali Linux or Parrot Security OS, prioritizing ease of installation, hardware requirements, and post-installation configuration for beginner and intermediate students. Evaluate and install the most suitable penetration testing operating system for a security team by exploring the installation processes and user-friendliness of Kali Linux and Parrot Security OS.<br>References:<br>1. Kali OS: https://www.kali.org/<br>2. Kali OS Installation: https://www.youtube.com/watch?v=jk2KGdJU2OI<br>3. Parrot Security OS: https://parrotlinux.org/<br>4. Parrot Security OS Installation: https://www.youtube.com/watch?v=4qvFp99rfXw | 02 | 1 |
| 2. | You are a penetration tester hired by a small corporate company to assess the security of their internal network. The network consists of several devices, including servers, routers, workstations, and IoT devices. The company has provided a range of IP addresses for you to scan, but they want you to assess the vulnerabilities and misconfigurations on their internal systems using Nmap.<br>Your objectives:<br>a. Map the network: Discover which devices are active on the network.<br>b. Identify open ports and services: Check which ports are open and identify the services running on them.<br>c. Identify OS and versions: Detect the operating systems and their versions running on different hosts.<br>d. Perform a vulnerability scan: Check for common vulnerabilities and exposures.<br>Reference: https://nmap.org/book/man.html | 02 | 2 |

| | | | |
|---|---|---|---|
| 3. | Languages exhibit distinct statistical characteristics, with certain letters appearing more frequently than others. For example, in English, the letter 'e' is the most common, followed by 't', 'a', and others. By comparing the frequency of characters in the encrypted text to this standard distribution, we can hypothesize which symbols correspond to which letters in the original language. Implement how statistical patterns in a language can assist in decoding an encrypted message. The encrypted text has been generated using a consistent letter-shifting method. Our objective is to analyze the frequency of characters within the text and uncover the underlying pattern to reconstruct the original message. | 02 | 3 |
| 4. | Encrypt a message using the standard Playfair cipher and attempt to decrypt it without the key by analyzing patterns in the ciphertext. Then, implement an extended Playfair cipher with a 10x10 matrix, incorporating uppercase/lowercase letters, digits, and symbols (e.g., @, #, $). Encrypt and decrypt a message with this extended cipher, demonstrating how the increased complexity improves security compared to the standard version. | 02 | 3 |
| 5. | XYZ Bank implements the encryption technique shown in figure to secure transactions between its servers and customers. The bank uses the following setup:<br>1. The server's public key is n=119 and e=5.<br>2. Customers encrypt sensitive information, such as their PINs, using this public key before sending it to the server.<br>A customer wants to send their 2-digit PIN, M=31, to the bank.<br>However, an attacker intercepts the encrypted message, which is C=92 and algorithm used. The attacker is determined to decrypt the ciphertext and discover the PIN. | 04 | 5 |

| | | | |
|---|---|---|---|
| |  <br>Implement the above scenario.<br>Reference: https://www.youtube.com/watch?v=VF3AHG0T9ec | | |
| 6. | Refer to the figure (a) attached here. Bob (Source A) is preparing to send a message to Alice (Destination B). Bob applies the SHA256 hash algorithm on the prepared message and appends it with original message (M) which is further encrypted by a single secret key. Alice will receive a bundle of encrypted H(M) and original messages (M). Alice will first apply a single secret key to decrypt the entire bundle and collect H(M) and the original message (M). Furthermore, Alice will apply the same algorithm SHA256 which was used by Bob, and produce a hash of the received message (H). Lastly, Alice will verify the computed hash with the received H(M) to make sure the message is not altered by any attackers.<br>  <br>Task to perform:<br>1. Use any Symmetric key/Asymmetric key algorithm to implement encryption function and decryption. | 04 | 5 |

| | | | |
|---|---|---|---|
| | 2. Implementation can be done using any programming language such as Java programming or python programming.<br>3. For SHA256 hashing, you may use library compatible as per your programming language.<br><br>Discuss the issues causes with this scenario. What happened if we encrypt the generated hash? | | |
| 7. | Refer to the attached figure here. Bob is preparing to send message to Alice. Bob and Alice both secretly computes the code(s) without sharing on any communication channel. Suggest key exchange algorithm to Bob and Alice for securely exchange informationwithoutsharing actual key.Once they formsecret code, Bob applies SHA256 hash algorithm on original message (M) plus code (s) and send hash of original message and code (M\|\|s) to Alice. Alice will receive bundle of H(M\|\|s) and first append code (s) with received message (M) and produce hash of the message (H) that compare with H(M\|\|s) to make sure that message is not altered by any attackers.<br>  <br><br>(c)<br>The task to perform:<br>1. Use some key exchange algorithm to calculate the value of s (secret code) which must be unique at the sender and receiver side.<br>2. Implementation can be done using any programming language such as Java programming or python programming.<br>3. Apply SHA256 on t h e message and secret code and display it on the output screen. Verify the hash value at the receiver end. | 04 | 6 |
| 8. | Show a practical scenario of Key Distribution. Use the separate key-sharing server that shares the secret key created using the AES-256-bit algorithm and share the secret key using the RSA algorithm with 1024/2048-bit key size. The key-sharing server produces a new secret key for each new communication between two nodes. | 02 | 5 |

| | | | |
|---|---|---|---|
| 9. | A digital signature is a mathematical scheme for presenting the authenticity of digital messages or documents. A valid digital signature gives a recipient reason to believe that<br>• The message was created by a claimed sender **(authentication)**,<br>• The sender cannot deny having sent the message **(non-repudiation)**,<br>• The message was not altered in transit.<br>Create a secure server that generates the digital certificate and shares it with the client machine. Show that information is signed and verified by the recipient entity to test the authenticity, non-repudiation, and integrity of the document (transaction). | 02 | 6 |
| 10. | A digital forensics team is investigating a case involving encrypted files and documents critical to their investigation. The team must recover the passwords for various applications, including archived files and PDF documents, using tools like Passware Password Recovery Kit Forensic, Advanced Archive Password Recovery, and Advanced PDF Password Recovery. Recover application passwords using specialized tools to demonstrate password recovery techniques and evaluate the efficiency of each tool in real-world scenarios.<br>References:<br>1. Passware Password Recovery Kit Forensic - https://www.passware.com/<br>2. Advanced Archive Password Recovery https://www.elcomsoft.com/archpr.html<br>3. Advanced PDF Password Recovery - https://www.elcomsoft.com/apdfpr.html | 02 | 2 |
| 11. | Study SQL Injection and cross-site scripting (XSS) and implement following practical scenario:<br>Set up an e-commerce web server with a login page and a product search feature. Test the server's security by attempting SQL Injection to bypass login authentication and XSS to inject malicious scripts into the product reviews section, demonstrating how attackers can exploit these vulnerabilities. | 04 | 2 |

# PRACTICAL: 1

## AIM:

A security training institute is setting up a lab for ethical hacking workshops. The team must decide whether to use Kali Linux or Parrot Security OS, prioritizing ease of installation, hardware requirements, and post-installation configuration for beginner and intermediate students. Evaluate and install the most suitable penetration testing operating system for a security team by exploring the installation processes and user-friendliness of Kali Linux and Parrot Security OS.

## THEORY:

**Penetration Testing OS Overview**: Kali Linux and Parrot Security OS are specialized Linux distributions for ethical hacking and penetration testing. Both come with a wide range of pre-installed security tools. Kali is more advanced, while Parrot offers a more beginner-friendly experience with privacy features.

**Ease of Installation & Configuration**: Kali Linux requires more manual configuration post-installation. Parrot OS is easier to set up and is optimized for lower-resource systems. Both distributions are used in cybersecurity training environments for hands-on experience.

### Tools in Kali Linux

**Nmap-** A network scanning tool for discovering hosts, open ports, and services running on a network, commonly used for vulnerability assessment and network mapping.

**Netcat-** A versatile networking tool used for reading and writing data across network connections. Often used for creating reverse shells and testing network connections.

**Fluxion-** A tool for performing social engineering-based Wi-Fi attacks, including phishing attacks to capture WPA handshakes and crack Wi-Fi passwords.

**Lynis-** A security auditing tool that performs system scans to detect vulnerabilities and weaknesses in a system's configuration, offering recommendations for hardening.

**Wireshark**- A network protocol analyzer used to capture and analyze network traffic in real-time, helpful for detecting issues like packet sniffing and man-in-the-middle attacks.

### Tools in Parrot Security OS

**Tor & Anonsurf-** Tor is used for anonymous browsing by routing traffic through multiple servers, while Anonsurf integrates Tor with Parrot OS to anonymize all network activity.

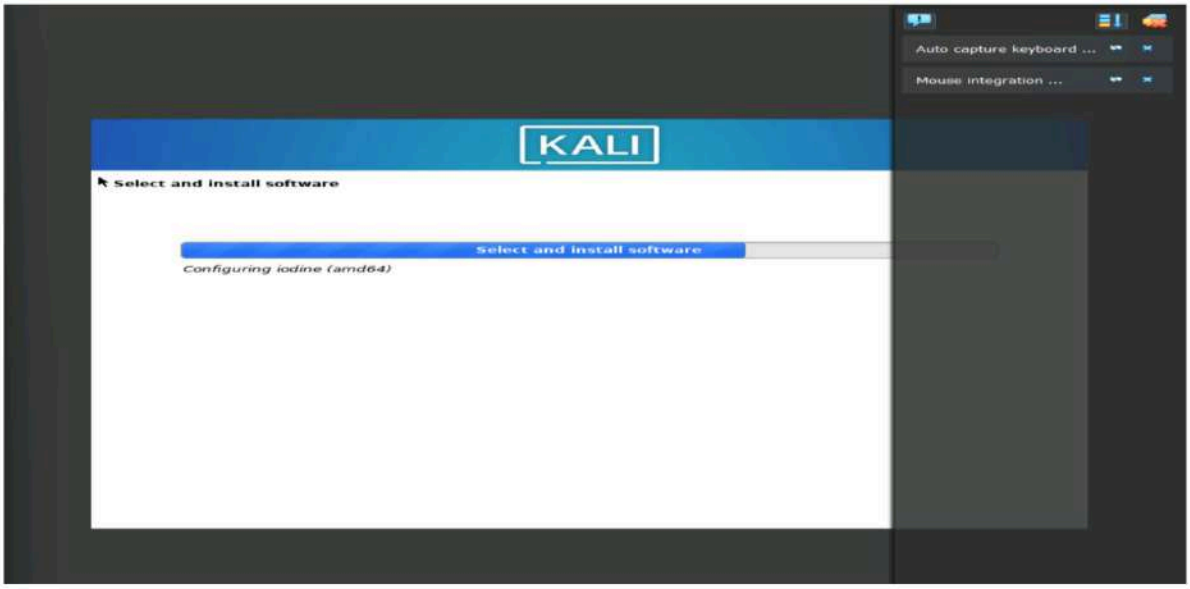
**Electrum-** A lightweight and secure Bitcoin wallet used for managing and transacting cryptocurrency with features like two-factor authentication and multi-signature support.

**Kayak-** A network monitoring tool that helps assess the security and performance of network traffic, similar to Wireshark, but with a lighter footprint and user-friendly interface.

**EtherApe-** A graphical network monitor that displays network activity in real-time. It visualizes connections and traffic flows between devices on a network.

**Ricochet-** A secure and anonymous instant messaging tool that uses the Tor network to allow users to communicate without revealing their location or identity.

## OUTPUT:




## LATEST APPLICATIONS:

**Latest Applications of Kali Linux**

**Penetration Testing-** Used to test networks, systems, and applications for vulnerabilities with pre-installed tools.

**Red Teaming-** Offensive teams use Kali for simulating attacks to assess security defenses.

**Wireless Network Auditing-** Tools like Aircrack-ng audit wireless networks for encryption weaknesses.

**Web Application Security Testing-** Tools like Burp Suite and OWASP ZAP identify web app vulnerabilities like SQL injection and XSS.

**Social Engineering-** Kali includes tools like Social-Engineer Toolkit (SET) for testing human vulnerabilities through phishing and other social engineering tactics.

**Exploit Development-** Kali provides environments and tools like Metasploit for creating and testing custom exploits on vulnerable systems.

**Latest Applications of Parrot Security OS**

**Privacy and Anonymity-** Tools like Tor and Anonsurf ensure anonymous browsing during testing.

**Digital Forensics-** Used for collecting and analyzing digital evidence in forensics investigations.

**IoT Security Testing-** Tools for testing the security of IoT devices and connected networks.

**Secure Communications-** PGP encryption and other tools ensure private communications during assessments.

**Cryptography-** Parrot offers tools for encryption and decryption, aiding in secure data handling and communication during tests.

**Cloud Security-** Parrot OS includes tools to test and secure cloud environments, identifying potential vulnerabilities in cloud infrastructure and services.

## LEARNING OUTCOME:

From this practical, I gained hands-on experience with Kali Linux and Parrot OS. I learned how to install, configure, and use both operating systems, which helped me develop problem-solving skills and understand how to choose the right OS for different tasks.

## REFERENCES:

1. Kali OS: https://www.kali.org/
2. Kali OS Installation: https://www.youtube.com/watch?v=jk2KGdJU2OI
3. Parrot Security OS: https://parrotlinux.org/
4. Parrot Security OS Installation: https://www.youtube.com/watch?v=4qvFp99rfXw
5. Kali Linux Tools: https://kali.org/tools/
6. Parrot Security Tools: https://linuxhint.com/parrot_os_tools_top_20/

# PRACTICAL: 2

## AIM:

You are a penetration tester hired by a small corporate company to assess the security of their internal network. The network consists of several devices, including servers, routers, workstations, and IoT devices. The company has provided a range of IP addresses for you to scan, but they want you to assess the vulnerabilities and misconfigurations on their internal systems using Nmap.
objectives:
1. Map the network: Discover which devices are active on the network.
2. Identify open ports and services: Check which ports are open and identify the services running on them.
3. Identify OS and versions: Detect the operating systems and their versions running on different hosts.
4. Perform a vulnerability scan: Check for common vulnerabilities and exposures.

## THEORY:

Network Scanning: A technique used to discover devices, services, and open ports in a network, essential for assessing security.
Nmap (Network Mapper): A powerful open-source tool for network discovery, service identification, OS detection, and vulnerability assessment.
Port Scanning: Identifies open ports and the services running on them, highlighting potential entry points for attackers.
OS Detection: Determines the operating system and version on target hosts to assess their compatibility and security posture.

## CODE:

### 1. Mapping and Scanning the Network

nmap -sn 192.168.57.174
-sn: Performs a ping scan to identify live hosts without scanning for open ports.

### 2. Identify the open ports and services

nmap -sS -sV 192.168.57.174
-sV: Enables version detection to identify services running on open ports.

### 3. Identify OS and versions

nmap -O 192.168.57.174
-O: Enables operating system detection.

### 4. Perform vulnerability scan

nmap --script vuln 192.168.57.174
--script vuln: Runs Nmap's built-in vulnerability scripts to check for known vulnerabilities.

**OUTPUT:**

```
File  Actions  Edit  View  Help
┌──(root㉿kali)-[~]
└─# sudo nmap -sn 192.168.238.94
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-01-03 09:17 IST
Nmap scan report for 192.168.238.94
Host is up (0.00071s latency).
Nmap done: 1 IP address (1 host up) scanned in 15.09 seconds
```

```
┌──(root㉿kali)-[~]
└─# nmap -sS -sV 192.168.238.94
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-01-03 09:40 IST
Nmap scan report for 192.168.238.94
Host is up (0.0016s latency).
All 1000 scanned ports on 192.168.238.94 are in ignored states.
Not shown: 1000 filtered tcp ports (no-response)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 20.27 seconds

┌──(root㉿kali)-[~]
└─# nmap -sA 192.168.238.94
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-01-03 09:41 IST
Stats: 0:00:03 elapsed; 0 hosts completed (0 up), 1 undergoing Ping Scan
Parallel DNS resolution of 1 host. Timing: About 0.00% done
Nmap scan report for 192.168.238.94
Host is up (0.00058s latency).
All 1000 scanned ports on 192.168.238.94 are in ignored states.
Not shown: 1000 unfiltered tcp ports (reset)

Nmap done: 1 IP address (1 host up) scanned in 13.91 seconds

┌──(root㉿kali)-[~]
└─#
```

```
┌──(root㉿kali)-[~]
└─# nmap -O 192.168.238.94
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-01-03 09:21 IST
Nmap scan report for 192.168.238.94
Host is up (0.0032s latency).
All 1000 scanned ports on 192.168.238.94 are in ignored states.
Not shown: 1000 filtered tcp ports (no-response)
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 clos
ed port
Aggressive OS guesses: 3Com 4500G switch (92%), H3C Comware 5.20 (92%), Huawei VRP 8.100 (92%)
, Microsoft Windows Server 2003 SP1 (92%), Oracle Virtualbox (92%), QEMU user mode network gat
eway (92%), AXIS 2100 Network Camera (92%), D-Link DP-300U, DP-G310, or Hamlet HPS01UU print s
erver (92%), HP Tru64 UNIX 5.1A (92%), Sanyo PLC-XU88 digital video projector (92%)
No exact OS matches for host (test conditions non-ideal).

OS detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 20.07 seconds
```

```
┌──(root㉿kali)-[~]
└─# nmap --script vuln 192.168.238.94
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-01-03 09:22 IST
Nmap scan report for 192.168.238.94
Host is up (0.0010s latency).
All 1000 scanned ports on 192.168.238.94 are in ignored states.
Not shown: 1000 filtered tcp ports (no-response)

Nmap done: 1 IP address (1 host up) scanned in 28.74 seconds

┌──(root㉿kali)-[~]
└─#
```

## LATEST APPLICATIONS:

Network Mapping and Inventory: Nmap creates detailed maps of computer networks, detects live hosts, and helps manage network inventories.

OS and Port Scanning: Identifies operating systems, open ports, and running services on local and remote systems for security assessments.

Vulnerability and Zero-Day Detection: Uses Nmap scripts to identify vulnerabilities, including zero-day threats, and enhance proactive security measures.

Web Application Testing: Assesses vulnerabilities in web servers and complements tools like Burp Suite for comprehensive analysis.

## LEARNING OUTCOME:

From this practical, I learned about network scanning and vulnerability assessment. I explored how to use Nmap to discover active devices and services on a network, identify open ports, detect operating systems running on different hosts, and perform basic vulnerability scans to highlight potential security risks.

## REFERENCES:

1. Nmap Guide: https://www.geeksforgeeks.org/nmap-command-in-linux-with-examples/
2. Nmap: https://nmap.org/book/man.html
3. Nmap Official: https://nmap.org/docs.html

# PRACTICAL: 3

## AIM:

Languages exhibit distinct statistical characteristics, with certain letters appearing more frequently than others. For example, in English, the letter 'e' is the most common, followed by 't', 'a', and others. By comparing the frequency of characters in the encrypted text to this standard distribution, we can hypothesize which symbols correspond to which letters in the original language. Implement how statistical patterns in a language can assist in decoding an encrypted message. The encrypted text has been generated using a consistent letter-shifting method. Our objective is to analyze the frequency of characters within the text and uncover the underlying pattern to reconstruct the original message.

## THEORY:

### Introduction to Frequency Analysis

Frequency analysis is a classical cryptographic technique used to break ciphers that involve substitution, such as the Caesar cipher or monoalphabetic substitution cipher. The core principle of frequency analysis relies on the fact that in natural languages, certain letters and patterns appear with a higher frequency than others. In the English language, for example, the letter 'e' is the most frequently occurring letter, followed by 't', 'a', and others. By analyzing the frequency of characters in an encrypted message (ciphertext), we can compare them to the typical letter distribution of the language and hypothesize which letters correspond to which in the original (plaintext) message.

### The Concept of Substitution Ciphers

In a substitution cipher, each letter in the plaintext is replaced by a different letter or symbol. The most basic substitution cipher is the Caesar cipher, where each letter is shifted by a fixed number in the alphabet. For example, with a shift of 3, A becomes D, B becomes E, and so on.

Substitution ciphers, especially simple ones like the Caesar cipher, are vulnerable to frequency analysis because the frequency of letters in the ciphertext follows the same distribution as in the plaintext. If we can identify patterns and match the most frequent letters in the ciphertext to the most frequent letters in the language, we can begin to reverse the encryption process.

### Why Frequency Analysis Works

Frequency analysis works because of the statistical properties of language. In English, certain letters, digraphs (pairs of letters), and trigraphs (three-letter combinations) appear more often than others. By leveraging these frequencies, we can gain insight into the structure of the encrypted message and begin to unravel the encryption.

Even in cases where the cipher uses a more complex substitution scheme, like the Vigenère cipher, frequency analysis can still be helpful, although it requires additional steps and may not work as directly as with simpler ciphers.

**Encryption Formula**
To encrypt a letter in the Caesar cipher, we use the following formula:

$C=(P+k) \bmod 26$

Where:

C = Ciphertext letter (encrypted letter)
P = Plaintext letter (original letter in the message)
k = Shift key (the number of positions each letter is shifted)
mod 26 = Ensures the result wraps around the alphabet (since there are 26 letters in the English alphabet)

**Decryption Formula**
To decrypt a letter, we reverse the shift (i.e., subtract the key instead of adding it):

$P=(C-k) \bmod 26$

Where:

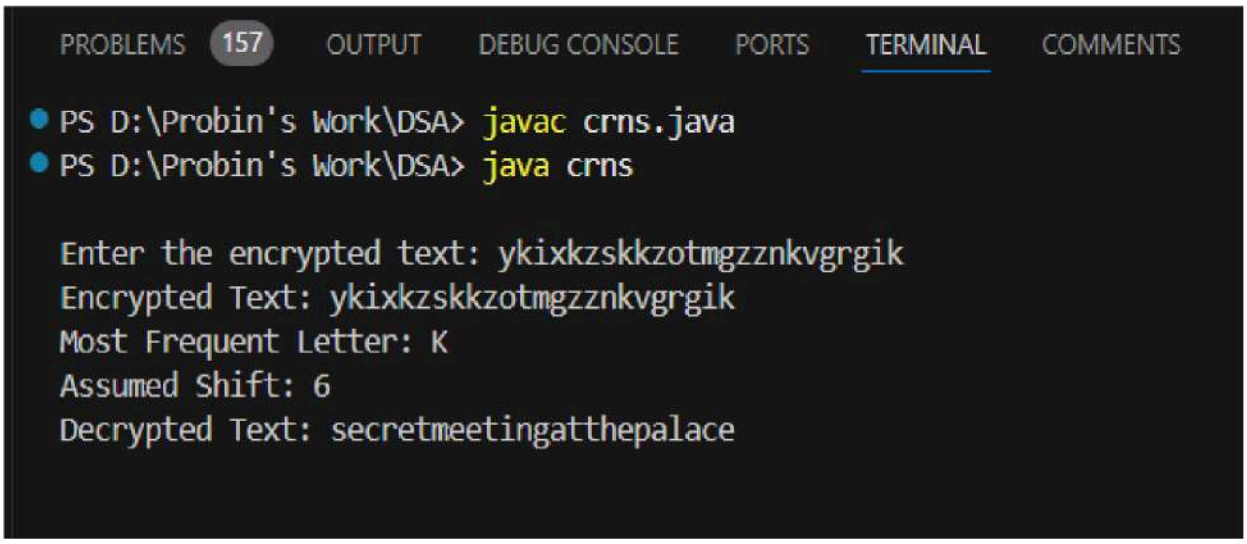
P = Plaintext letter (decrypted letter)
C = Ciphertext letter (encrypted letter)
k = Shift key (the same key used during encryption)
mod 26 = Ensures the result wraps around the alphabet

## CODE:

```
import java.util.Scanner;

public class crns {

    public static String decryptCaesar(String text, int shift) {
        StringBuilder decrypted = new StringBuilder();

        for (char c : text.toCharArray()) {
            if (Character.isAlphabetic(c)) {
                char base = Character.isUpperCase(c) ? 'A' : 'a';
                decrypted.append((char) (((c - base - shift + 26) % 26) + base));
            } else {
                decrypted.append(c);
            }
        }

        return decrypted.toString();
    }

    public static char findMostFrequentLetter(String text) {
        int[] frequency = new int[26];
```

```
        for (char c : text.toCharArray()) {
            if (Character.isAlphabetic(c)) {
                char upperChar = Character.toUpperCase(c);
                frequency[upperChar - 'A']++;
            }
        }

        int maxFreq = 0;
        char mostFrequent = ' ';
        for (int i = 0; i < 26; i++) {
            if (frequency[i] > maxFreq) {
                maxFreq = frequency[i];
                mostFrequent = (char) ('A' + i);
            }
        }

        return mostFrequent;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("\nEnter the encrypted text: ");
        String encryptedText = scanner.nextLine();

        System.out.println("Encrypted Text: " + encryptedText);

        char mostFrequent = findMostFrequentLetter(encryptedText);
        System.out.println("Most Frequent Letter: " + mostFrequent);

        int shift = (mostFrequent - 'E' + 26) % 26;
        System.out.println("Assumed Shift: " + shift);

        String decryptedText = decryptCaesar(encryptedText, shift);
        System.out.println("Decrypted Text: " + decryptedText);
        System.out.println("\n\n");

        scanner.close();
    }
}
```

## OUTPUT:




## LATEST APPLICATIONS:

### 1. Cryptography and Cybersecurity

Password Strength Analysis: Frequency analysis helps evaluate password strength by identifying common patterns and character combinations used in weak passwords. It assists in creating better hashing algorithms to defend against brute-force attacks.

Breaking Weak Encryption: Older encryption methods, like substitution ciphers, can still be broken using frequency analysis by exploiting predictable letter distributions.

### 2. Data Compression

Huffman Coding: Frequency analysis is central to Huffman coding, a data compression technique that assigns shorter binary codes to frequently occurring characters, reducing file sizes.

Run-Length Encoding: This technique uses frequency analysis to efficiently compress data by encoding repeated patterns, like consecutive identical characters in an image.

### 3. Natural Language Processing (NLP)

Language Identification: Frequency analysis helps identify the language of a given text by comparing its letter frequency distribution with known language profiles.

Spell Checkers and Autocorrect: Statistical patterns of language are applied to detect and correct typographical errors based on common letter combinations.

### 4. Digital Forensics

Decoding Hidden Messages: Frequency analysis helps investigators decrypt messages hidden in digital media or communication, particularly in cases of steganography. Recovering Deleted or Damaged Ciphertext: It assists in recovering compromised or deleted encrypted messages by analyzing partial ciphertext data.

## LEARNING OUTCOME:

1. **Understanding Cryptography**: I learned how classical ciphers like the Caesar cipher work and how they can be broken using statistical analysis.

2. **Hands-on Cryptanalysis**: The practical helped me understand how cryptanalysis techniques, such as frequency analysis, are applied in real-world cryptographic attacks.

3. **Critical Thinking**: The practical helped me develop problem-solving skills by analyzing and decrypting a cipher systematically.

## REFERENCES:

1. Tutorialspoint: https://www.tutorialspoint.com/cryptography/index.htm

2. GeeksforGeeks: https://www.geeksforgeeks.org/caesar-cipher-in-cryptography

3. Cryptography: https://www.khanacademy.org/computing/computer-science/cryptography

# PRACTICAL: 4

## AIM:

Encrypt a message using the standard Playfair cipher and attempt to decrypt it without the key by analyzing patterns in the ciphertext. Then, implement an extended Playfair cipher with a 10x10 matrix, incorporating uppercase/lowercase letters, digits, and symbols (e.g., @, #, $). Encrypt and decrypt a message with this extended cipher, demonstrating how the increased complexity improves security compared to the standard version.

## THEORY:

Playfair Cipher

The Playfair cipher is a digraph substitution cipher that encrypts pairs of letters instead of single letters, making it more secure than simple monoalphabetic ciphers. It uses a 5x5 matrix of letters, constructed using a keyword, and follows specific rules for encryption and decryption.

Encryption Rules:

1. If both letters in a pair appear in the same row, replace each with the letter to its right (wrapping around if needed).
2. If both letters are in the same column, replace each with the letter below it (wrapping around if needed).
3. If the letters form a rectangle, replace them with the letters on the same row but in the opposite corners.
4. If the plaintext contains repeated letters in a pair, an 'X' is inserted between them.
5. If the number of characters is odd, an extra 'X' is added at the end.

## CODE:

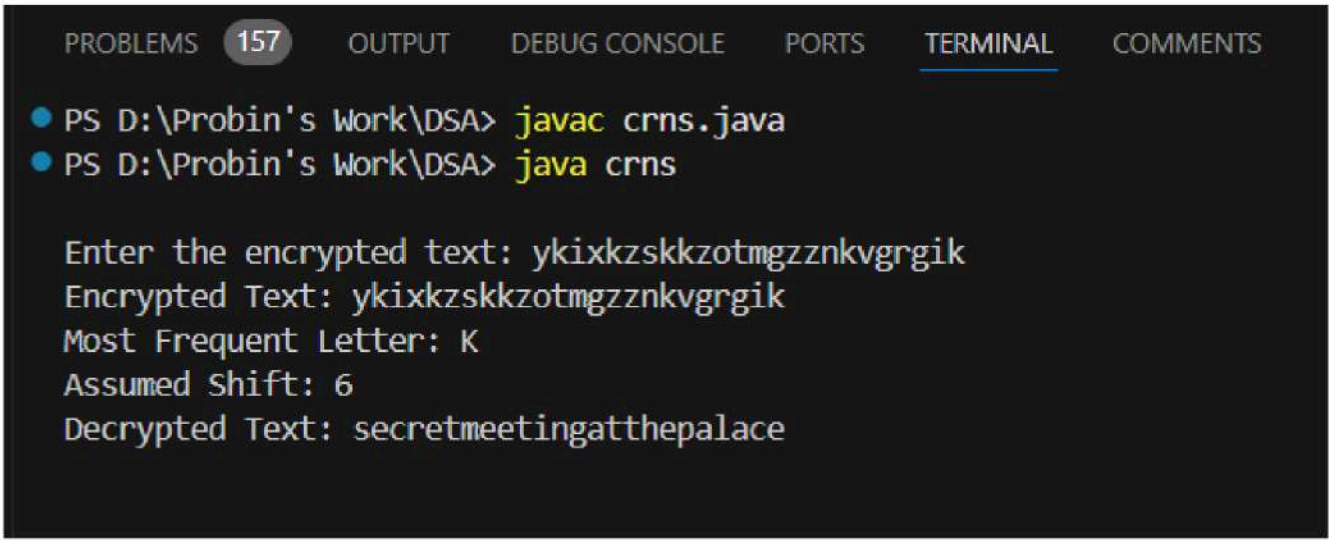
```
import java.util.*;

public class crns {

   private static char[][] keyMatrix = new char[5][5];

   public static void main(String[] args) {

      Scanner scanner = new Scanner(System.in);



      System.out.print("Enter the key (without spaces): ");

      String key = scanner.nextLine().toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");

      System.out.print("Enter the plaintext (without spaces): ");

      String plaintext = scanner.nextLine().toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");
```

```
        generateKeyMatrix(key);
        displayMatrix();
        String encryptedText = encryptText(plaintext);
        System.out.println("Encrypted Text: " + encryptedText);
        scanner.close();
    }
    private static void generateKeyMatrix(String key) {
        StringBuilder uniqueKey = new StringBuilder();
        boolean[] used = new boolean[26];
        for (char c : key.toCharArray()) {
            if (!used[c - 'A'] && c != 'J') {
                uniqueKey.append(c);
                used[c - 'A'] = true;
            }
        }
    for (char c = 'A'; c <= 'Z'; c++) {
            if (!used[c - 'A'] && c != 'J') {
                uniqueKey.append(c);
            }
        }
        int index = 0;
        for (int row = 0; row < 5; row++) {
            for (int col = 0; col < 5; col++) {
                keyMatrix[row][col] = uniqueKey.charAt(index++);
            }
        }
    }
    private static void displayMatrix() {
        System.out.println("Key Matrix:");
        for (int row = 0; row < 5; row++) {
```

```
            for (int col = 0; col < 5; col++) {
                System.out.print(keyMatrix[row][col] + " ");
            }
            System.out.println();
        }
    }
    private static String encryptText(String text) {
        StringBuilder preparedText = prepareText(text);
        StringBuilder encryptedText = new StringBuilder();
        for (int i = 0; i < preparedText.length(); i += 2) {
            char first = preparedText.charAt(i);
            char second = preparedText.charAt(i + 1);
            int[] pos1 = findPosition(first);
            int[] pos2 = findPosition(second);
            if (pos1[0] == pos2[0]) { // Same row
                encryptedText.append(keyMatrix[pos1[0]][(pos1[1] + 1) % 5]);
                encryptedText.append(keyMatrix[pos2[0]][(pos2[1] + 1) % 5]);
            } else if (pos1[1] == pos2[1]) { // Same column
                encryptedText.append(keyMatrix[(pos1[0] + 1) % 5][pos1[1]]);
                encryptedText.append(keyMatrix[(pos2[0] + 1) % 5][pos2[1]]);
            } else { // Rectangle swap
                encryptedText.append(keyMatrix[pos1[0]][pos2[1]]);
                encryptedText.append(keyMatrix[pos2[0]][pos1[1]]);
            }
        }
        return encryptedText.toString();
    }
    private static StringBuilder prepareText(String text) {
        StringBuilder preparedText = new StringBuilder(text);
        for (int i = 0; i < preparedText.length() - 1; i += 2) {
```

```
            if (preparedText.charAt(i) == preparedText.charAt(i + 1)) {
                preparedText.insert(i + 1, 'X');
            }
        }
        if (preparedText.length() % 2 != 0) {
            preparedText.append('X');
        }
        return preparedText;
    }
    private static int[] findPosition(char c) {
        for (int row = 0; row < 5; row++) {
            for (int col = 0; col < 5; col++) {
                if (keyMatrix[row][col] == c) {
                    return new int[]{row, col};
                }
            }
        }
        return null;
    }
}
```

## OUTPUT:



## LATEST APPLICATIONS:

- **Secure Communications –** Used in low-resource encryption systems, such as military field encryption and emergency communications.
- **Embedded Systems Security –** Applied in IoT devices for lightweight cryptographic solutions.
- **Data Integrity Checks –** Helps in verifying message integrity in constrained environments.
- **Educational Cryptanalysis –** A useful tool in cybersecurity training for understanding classical cryptographic weaknesses and improvements.

## LEARNING OUTCOME:

- Practical Application of Cryptography: Developed encryption and decryption programs in Java, reinforcing concepts of digraph encryption methods
- Understand the working of the Playfair cipher for encryption and decryption.
- Analyze ciphertext to identify patterns and weaknesses in traditional ciphers.
- Implement an extended Playfair cipher to enhance security using a larger character set.

## References:

1. https://www.baeldung.com/cs/playfair-cipher
2. https://www.tutorialspoint.com/cryptography/cryptography_playfair_cipher.htm
3. https://www.geeksforgeeks.org/playfair-cipher-with-examples/

# PRACTICAL: 5

## AIM:

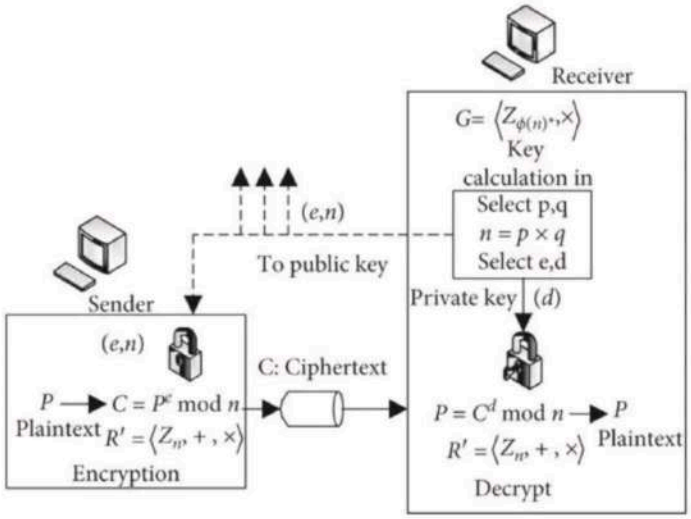
XYZ Bank implements the encryption technique shown in figure to secure transactions between its servers and customers. The bank uses the following setup:

1. The server's public key is n=119 and e=5.

2. Customers encrypt sensitive information, such as their PINs, using this public key before sending it to the server.

A customer wants to send their 2-digit PIN, M=31, to the bank.

However, an attacker intercepts the encrypted message, which is C=92 and algorithm used. The attacker is determined to decrypt the ciphertext and discover the PIN.




Implement the above scenario.

## THEORY:

### Key Generation

Choose two prime numbers p and q.

Compute $\boldsymbol{n = p \times q}$

Calculate $\boldsymbol{\emptyset(n) = (p-1) \times (q-1)}$.

### Encryption

The plaintext M is encrypted using the public key:

## CODE:

```
import java.util.*;
 public class crns{

    public static long modExp(long base, long exp, long mod) {
       long result = 1;
       while (exp > 0) {
          if (exp % 2 == 1) {
             result = (result * base) % mod;
          }
          base = (base * base) % mod;
          exp /= 2;
       }
       return result;
    }

    public static long modInverse(long e, long phi) {
       for (long d = 1; d < phi; d++) {
          if ((e * d) % phi == 1) {
             return d;
          }
       }
       return -1;
    }

    public static void main(String[] args) {
       long p = 9, q = 18;
       long n = p * q;
       long e = 5;
       long phi = (p - 1) * (q - 1);
       long d = modInverse(e, phi);
       if (d == -1) {
          System.err.println("Error computing modular inverse. Exiting.");
          return;
       }

       long M = 31;
       long C = modExp(M, e, n);
       System.out.println("Encrypted PIN (C): " + C);


       long intercepted_C = 92;
       long decrypted_M = modExp(intercepted_C, d, n);
       System.out.println("Decrypted PIN (M): " + decrypted_M);
    }
}
```

## OUTPUT:

```
PS D:\Probin's Work\DSA> javac alu.java
PS D:\Probin's Work\DSA> java alu
Encrypted PIN (C): 25
Decrypted PIN (M): 92
PS D:\Probin's Work\DSA>
```

## LATEST APPLICATIONS:

1. Secure Web Browsing (HTTPS): RSA is widely used in SSL/TLS protocols to ensure secure communication over the internet. It's used for key exchange during the handshake process in HTTPS, allowing web servers and browsers to securely share encryption keys.

2. Digital Signatures: RSA is commonly used in digital signatures to authenticate the identity of the sender and verify the integrity of messages. It's used in email services, software distribution, and legal documents to ensure that a message or file hasn't been altered.
3. VPNs and Secure Communication Channels: RSA is used in Virtual Private Networks (VPNs) to encrypt the data transferred between users and networks, ensuring that sensitive data remains secure even over insecure networks like the internet.

4. Cloud Security: In cloud computing environments, RSA is utilized for securing data transfer between clients and cloud servers. It's also used in data storage encryption to prevent unauthorized access to sensitive data stored on cloud platforms.

## LEARNING OUTCOME:

- Learn RSA Algorithm – Explore key generation, encryption, and decryption using modular arithmetic.
- Develop Cryptographic Functions – Implement modular exponentiation and modular inverse in Java.
- Expand Cybersecurity Expertise – Understand data protection strategies in practical applications.

## REFERENCES:

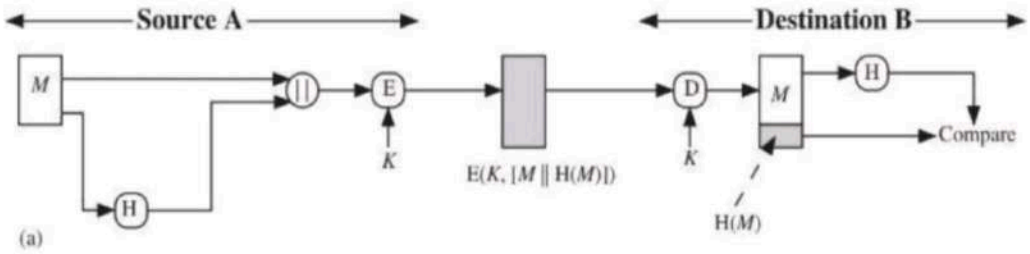
1. https://www.geeksforgeeks.org/rsa-algorithm-cryptography/

2. https://www.youtube.com/watch?v=VF3AHG0T9ec

# PRACTICAL: 6

## AIM:

Refer to the figure (a) attached here. Bob (Source A) is preparing to send a message to Alice (Destination B). Bob applies the SHA256 hash algorithm on the prepared message and appends it with original message (M) which is further encrypted by a single secret key. Alice will receive a bundle of encrypted H(M) and original messages (M). Alice will first apply a single secret key to decrypt the entire bundle and collect H(M) and the original message (M). Furthermore, Alice will apply the same algorithm SHA256 which was used by Bob, and produce a hash of the received message (H). Lastly, Alice will verify the computed hash with the received H(M) to make sure the message is not altered by any attackers.




Task to perform:

1. Use any Symmetric key/Asymmetric key algorithm to implement encryption function and decryption.

2. Implementation can be done using any programming language such as Java programming or python programming.

3. For SHA256 hashing, you may use library compatible as per your programming language.

## THEORY:

In cryptography, ensuring data integrity and confidentiality is crucial. SHA-256 is a cryptographic hash function that takes an input of any size and produces a fixed 256-bit output. It is a one-way function, meaning it cannot be reversed, making it ideal for verifying data integrity, ensuring that the message hasn't been altered during transmission.

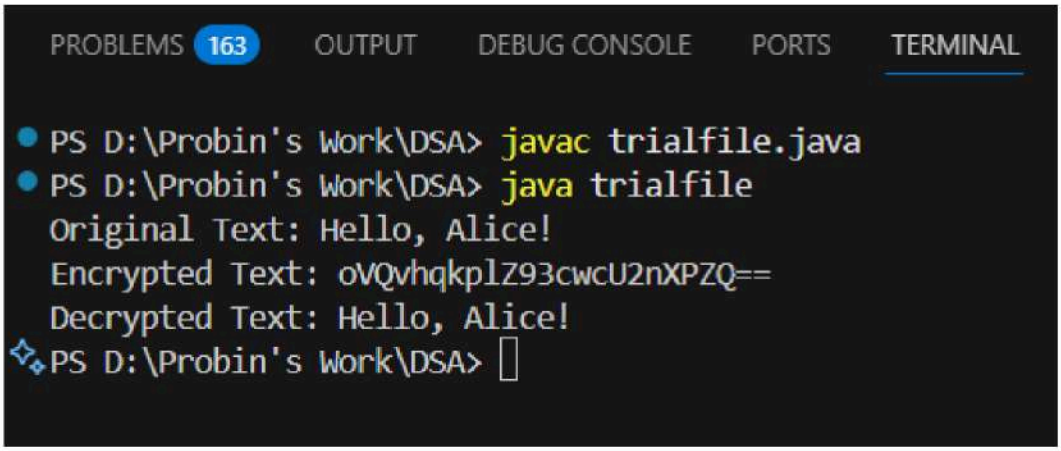
AES (Advanced Encryption Standard) is a symmetric encryption algorithm used to secure data by encrypting it with a secret key. AES uses the same key for both encryption and decryption and supports key sizes of 128, 192, or 256 bits. It's widely used for securing communication, data storage, and more.

In this practical, SHA-256 is used to generate a hash of the original message to check its integrity, while AES encrypts the message and the hash for confidentiality. The same secret key is used to decrypt the bundle and verify the message integrity, ensuring secure communication.

## CODE:

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.util.Base64;

public class trialfile {
    public static void main(String[] args) throws Exception {
        //AES key (128-bit)
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(128);
        SecretKey secretKey = keyGen.generateKey();

        // Plaintext
        String plaintext = "Hello, Alice!";
        System.out.println("Original Text: " + plaintext);

        // Encryption
        String encryptedText = encrypt(plaintext, secretKey);
        System.out.println("Encrypted Text: " + encryptedText);

        // Decryption
        String decryptedText = decrypt(encryptedText, secretKey);
        System.out.println("Decrypted Text: " + decryptedText);
    }

    // AES Encryption
    public static String encrypt(String plaintext, SecretKey secretKey) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes());
        return Base64.getEncoder().encodeToString(encryptedBytes);
    }

    // AES Decryption
    public static String decrypt(String encryptedText, SecretKey secretKey) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(encryptedText));
        return new String(decryptedBytes);
    }
}
```

## OUTPUT:




## LATEST APPLICATIONS:

1. **Secure Web Browsing (HTTPS):**
   SHA-256 is commonly used in SSL/TLS protocols, where it plays a critical role in creating secure connections between web browsers and servers. It is used in the creation of digital certificates, ensuring the integrity and authenticity of websites. This allows users to trust that the websites they are visiting have not been tampered with.
2. **Digital Signatures and Certificates:**
   SHA-256 is widely used in digital signatures for verifying the integrity of messages and documents. It ensures that data has not been altered during transmission. Popular services, including email authentication, software updates, and legal documents, rely on SHA-256 for generating unique signatures that represent the data.
3. **Cryptocurrency and Blockchain Technology:**
   SHA-256 is fundamental to the functioning of blockchain and cryptocurrency, particularly Bitcoin. It is used in the process of mining and generating blocks, ensuring that transactions are secure and that the blockchain remains tamper-proof. SHA-256's one-way hashing function ensures that data is securely stored and verified within the blockchain.
4. **Password Storage and Authentication:**
   In modern systems, SHA-256 is used to securely hash passwords before they are stored in databases. It ensures that even if a database is compromised, the actual passwords remain safe, as only the hashed values are stored. Many secure authentication mechanisms use SHA-256 hashing for creating strong password verification processes.

## LEARNING OUTCOME:

1. **Understand SHA-256 Algorithm:** Learn how SHA-256 generates a fixed-size hash from variable input and its role in ensuring data integrity.

2. **Apply AES Encryption:** Implement AES encryption and decryption with a secret key to secure data, while using SHA-256 to verify message integrity.

3. **Develop Secure Communication Systems:** Gain experience in applying encryption (AES) and hashing (SHA-256) to create secure communication systems.

4. **Understand Key Management in Cryptography:** Recognize the importance of key management in symmetric encryption for maintaining security.

5. **Enhance Cryptography Skills:** Develop practical skills in cryptography, including hashing and encryption, to secure data effectively.

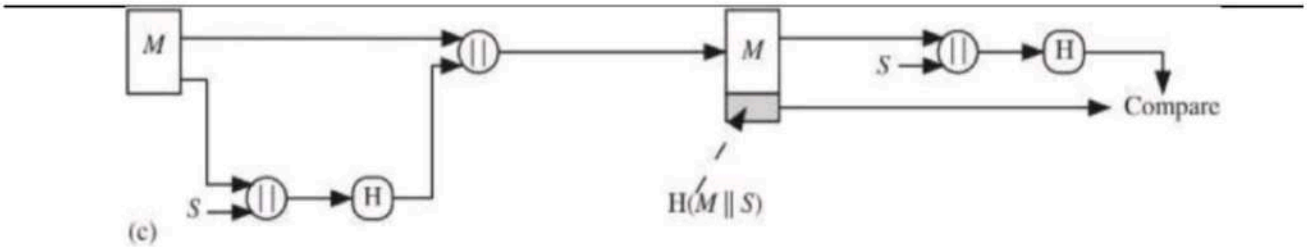
## REFERENCES:

1. https://www.geeksforgeeks.org/sha-256-in-cryptography/
2. https://www.tutorialspoint.com/cryptography/aes_encryption.htm
3. https://www.geeksforgeeks.org/sha-256-hash-in-java/
4. https://www.openssl.org/docs/manmaster/man7/AES.html

# PRACTICAL: 7

## AIM:

Refer to the attached figure here. Bob is preparing to send message to Alice. Bob and Alice both secretly computes the code(s) without sharing on any communication channel. Suggest key exchange algorithm to Bob and Alice for securely exchange information without sharing actual key. Once they form secret code, Bob applies SHA256 hash algorithm on original message (M) plus code (s) and send hash of original message and code (M||s) to Alice. Alice will receive bundle of H(M||s) and first append code (s) with received message (M) and produce hash of the message (H) that compare with H(M||s) to make sure that message is not altered by any attackers.




The task to perform:

1. Use some key exchange algorithm to calculate the value of s (secret code) which must be unique at the sender and receiver side.

2. Implementation can be done using any programming language such as Java programming or python programming.

3. Apply SHA256 on t h e message and secret code and display it on the output screen. Verify the hash value at the receiver end.

## THEORY:

Key exchange algorithms allow two parties to securely establish a shared secret without directly transmitting the key. One of the most widely used key exchange algorithms is Diffie-Hellman (DH) Key Exchange, which enables Bob and Alice to generate a unique secret key (s) without exposure.

In this practical, Bob and Alice independently generate the secret code (s) using a key exchange algorithm. Once established, Bob applies the SHA-256 hashing algorithm to the concatenated message and secret code (M||s) and transmits the hash to Alice. Alice then verifies the integrity of the received message by computing H(M||s) and comparing it with Bob's transmitted hash.

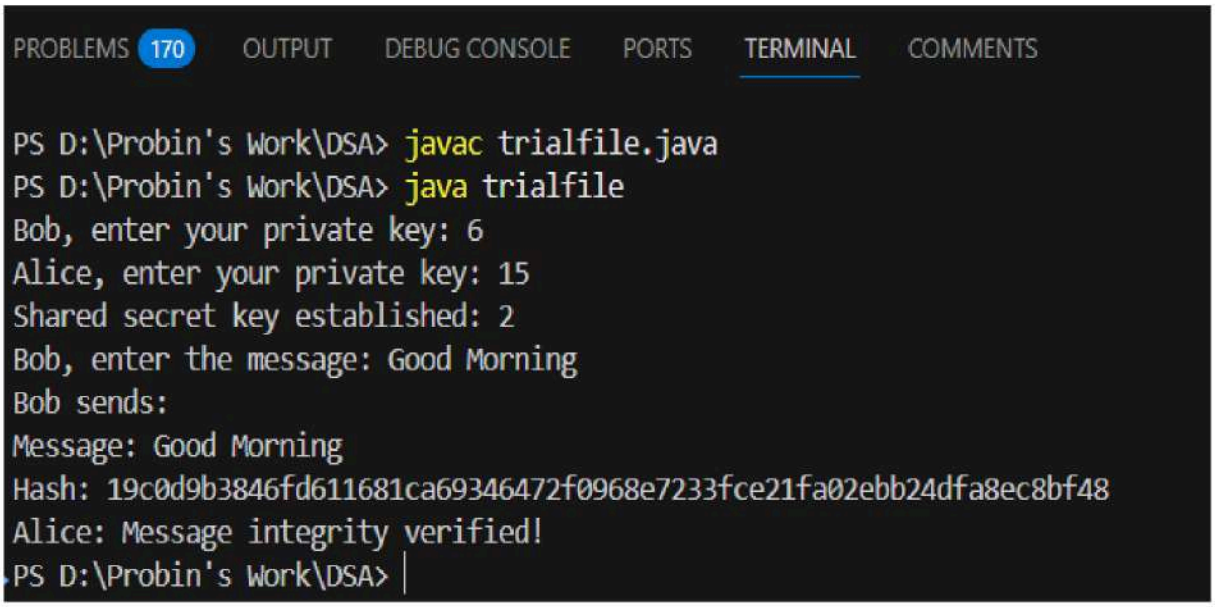
SHA-256 is a cryptographic hash function that ensures data integrity and prevents unauthorized modifications. The received hash is validated to check whether the message was altered during transmission.

## CODE:

```
import java.util.*;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Scanner;

public class trialfile {

   public static final BigInteger P = new BigInteger("23");
   public static final BigInteger G = new BigInteger("5");

   public static void main(String[] args) throws NoSuchAlgorithmException {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Bob, enter your private key: ");
      BigInteger bobPrivateKey = scanner.nextBigInteger();

      System.out.print("Alice, enter your private key: ");
      BigInteger alicePrivateKey = scanner.nextBigInteger();

      BigInteger bobPublicKey = G.modPow(bobPrivateKey, P);
      BigInteger alicePublicKey = G.modPow(alicePrivateKey, P);

      BigInteger bobSharedSecret = alicePublicKey.modPow(bobPrivateKey, P);
      BigInteger aliceSharedSecret = bobPublicKey.modPow(alicePrivateKey, P);

      if (bobSharedSecret.equals(aliceSharedSecret)) {
         System.out.println("Shared secret key established: " + bobSharedSecret);
      } else {
         System.out.println("Error in key exchange!");
         return;
      }

      System.out.print("Bob, enter the message: ");
      scanner.nextLine();
      String message = scanner.nextLine();

      String messageWithSecret = message + bobSharedSecret;
      String hash = sha256(messageWithSecret);

      System.out.println("Bob sends:");
      System.out.println("Message: " + message);
      System.out.println("Hash: " + hash);
```

```
        String receivedMessage = message;
        String verifyHash = sha256(receivedMessage + aliceSharedSecret);

        if (verifyHash.equals(hash)) {
            System.out.println("Alice: Message integrity verified!");
        } else {
            System.out.println("Alice: Message was altered!");
        }

        scanner.close();
    }

    // SHA-256 Hash Function
    public static String sha256(String input) throws NoSuchAlgorithmException {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hashBytes = digest.digest(input.getBytes());
        StringBuilder hexString = new StringBuilder();
        for (byte b : hashBytes) {
            hexString.append(String.format("%02x", b));
        }
        return hexString.toString();
    }
}
```

## OUTPUT:



## LATEST APPLICATIONS:

1. **Secure Online Transactions** – Used in digital banking and payment systems to verify transaction integrity. This prevents unauthorized modifications to financial data, ensuring safe and tamper-proof transactions.

2. **Digital Signatures** – Ensures data authenticity and prevents tampering in emails and legal documents. Businesses and government agencies use this technique to verify the legitimacy of digital contracts and sensitive communications.

3. **Blockchain Security** – Hashing algorithms like SHA-256 secure transactions in cryptocurrencies like Bitcoin. This ensures that once a block is added to the blockchain, its contents cannot be altered, making transactions immutable.

## LEARNING OUTCOME:

1. Understand Key Exchange Algorithms– I understood how Diffie-Hellman securely establishes a shared secret between two parties which is essential for designing secure communication systems in real-world applications.

2. Implement SHA-256 for Message Integrity– I applied cryptographic hashing to verify that data remains unaltered during transmission which helps in securing sensitive data, ensuring that the transmitted information remains intact and unchanged.

3. Enhance Cybersecurity Skills– Gained insights into real-world secure communication techniques by implementing encryption and hashing methods. This helped me understand how to protect data from cyber threats and unauthorized modifications.

## REFERENCES:

1. https://www.geeksforgeeks.org/implementation-diffie-hellman-algorithm/
2. https://www.techtarget.com/searchsecurity/definition/Diffie-Hellman-key-exchange
3. https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm

# PRACTICAL: 8

## AIM:

Show a practical scenario of Key Distribution. Use the separate key-sharing server that shares the secret key created using the AES-256-bit algorithm and share the secret key using the RSA algorithm with 1024/2048-bit key size. The key-sharing server produces a new secret key for each new communication between two nodes.

## THEORY:

Key distribution is a crucial element of cryptography, focusing on how secret keys are safely shared between parties. In this case, a hybrid cryptographic system combines the strengths of symmetric encryption (like AES) and asymmetric encryption (like RSA). The process begins with a trusted third party, often a key-sharing server, which generates secure, random AES-256 session keys. These keys are used to encrypt the communication data, offering strong security and efficiency. However, since these symmetric keys need to be exchanged securely, the system utilizes RSA public-key encryption to distribute them.

To distribute the AES session key, the server encrypts it with the public RSA key of each recipient, ensuring that only the intended recipient can decrypt the session key using their private RSA key. This method allows for secure key exchange without needing a pre-existing shared secret between the parties. By using this approach, the system ensures that the AES session key is safely delivered to the recipient while leveraging the benefits of asymmetric encryption for key distribution.

Each time a new communication session occurs, the server generates a fresh AES key, ensuring perfect forward secrecy. This means that even if one session key is compromised, the data from other sessions remains protected. This hybrid cryptographic system effectively combines the computational efficiency of symmetric encryption for data transmission with the secure key distribution of asymmetric encryption. The result is a secure and practical system well-suited for real-world cryptographic applications.

## Code:

```
import os
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
import base64
import time

class Node:
    def __init__(self, name):
        self.name = name
        # Generate RSA key pair for this node
        self.private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
        )
        self.public_key = self.private_key.public_key()
        self.session_keys = {}  # Store session keys for different communications
```

```
    def get_public_key_pem(self):
        """Return public key in PEM format"""
        return self.public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )

    def decrypt_session_key(self, encrypted_key):
        """Decrypt a session key that was encrypted with this node's public key"""
        decrypted_key = self.private_key.decrypt(
            encrypted_key,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return decrypted_key

    def encrypt_message(self, message, other_node_name):
        """Encrypt a message using the session key for communication with other_node"""
        if other_node_name not in self.session_keys:
            raise ValueError(f"No session key established with {other_node_name}")

        session_key = self.session_keys[other_node_name]

        # Generate a random IV
        iv = os.urandom(16)

        # Encrypt with AES-256 in CBC mode
        encryptor = Cipher(
            algorithms.AES(session_key),
            modes.CBC(iv)
        ).encryptor()

        # Pad the message to be a multiple of 16 bytes
        padded_message = message + b' ' * (16 - len(message) % 16)
        ciphertext = encryptor.update(padded_message) + encryptor.finalize()

        # Return IV + ciphertext
        return iv + ciphertext

    def decrypt_message(self, encrypted_data, other_node_name):
        """Decrypt a message using the session key for communication with other_node"""
        if other_node_name not in self.session_keys:
            raise ValueError(f"No session key established with {other_node_name}")

        session_key = self.session_keys[other_node_name]

        # Extract IV (first 16 bytes)
```

```
        iv = encrypted_data[:16]
        ciphertext = encrypted_data[16:]

        # Decrypt with AES-256 in CBC mode
        decryptor = Cipher(
            algorithms.AES(session_key),
            modes.CBC(iv)
        ).decryptor()

        plaintext = decryptor.update(ciphertext) + decryptor.finalize()
        return plaintext.rstrip()  # Remove padding

class KeySharingServer:
    def __init__(self):
        self.node_public_keys = {}  # Store public keys of registered nodes
        self.session_counter = 0

    def register_node(self, node_name, public_key_pem):
        """Register a node's public key with the server"""
        self.node_public_keys[node_name] = serialization.load_pem_public_key(public_key_pem)
        print(f"Node {node_name} registered with the key server")

    def create_secure_channel(self, node1_name, node2_name):
        """Generate a new AES-256 session key and distribute it to both nodes securely"""
        if node1_name not in self.node_public_keys or node2_name not in self.node_public_keys:
            raise ValueError("Both nodes must be registered with the key server")

        # Generate a new AES-256 key
        session_key = os.urandom(32)  # 256 bits = 32 bytes
        self.session_counter += 1
        session_id = f"session_{self.session_counter}_{int(time.time())}"

        # Encrypt the session key with each node's public key
        encrypted_key_node1 = self.node_public_keys[node1_name].encrypt(
            session_key,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )

        encrypted_key_node2 = self.node_public_keys[node2_name].encrypt(
            session_key,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
```

```
        print(f"New AES-256 session key generated for communication between {node1_name} and
{node2_name}")

        # Return the encrypted keys to be distributed to each node
        return {
            "session_id": session_id,
            node1_name: encrypted_key_node1,
            node2_name: encrypted_key_node2
        }

# Simulation of the key distribution scenario
def simulate_key_distribution():
    # Create a key sharing server
    key_server = KeySharingServer()

    # Create two nodes that want to communicate
    node_a = Node("Node_A")
    node_b = Node("Node_B")

    # Register nodes with the key server
    key_server.register_node(node_a.name, node_a.get_public_key_pem())
    key_server.register_node(node_b.name, node_b.get_public_key_pem())

    # Key server creates a secure channel by generating and distributing a session key
    encrypted_keys = key_server.create_secure_channel(node_a.name, node_b.name)

    # Nodes receive their encrypted session keys and decrypt them
    node_a.session_keys[node_b.name] = node_a.decrypt_session_key(encrypted_keys[node_a.name])
    node_b.session_keys[node_a.name] = node_b.decrypt_session_key(encrypted_keys[node_b.name])

    print(f"Session established: {encrypted_keys['session_id']}")

    # Now nodes can communicate securely using the shared session key
    message = b"Hello, this is a secure message using our shared AES-256 key."
    encrypted_message = node_a.encrypt_message(message, node_b.name)

    print(f"\nNode A encrypts: {message.decode()}")
    print(f"Encrypted data size: {len(encrypted_message)} bytes")

    # Node B receives and decrypts the message
    decrypted_message = node_b.decrypt_message(encrypted_message, node_a.name)
    print(f"Node B decrypts: {decrypted_message.decode()}")

    # For a new communication session, a new key is generated
    print("\n--- Starting new communication session ---")
    new_encrypted_keys = key_server.create_secure_channel(node_a.name, node_b.name)

    # Nodes receive their new encrypted session keys and decrypt them
    node_a.session_keys[node_b.name] =
node_a.decrypt_session_key(new_encrypted_keys[node_a.name])
```

```
    node_b.session_keys[node_a.name] =
node_b.decrypt_session_key(new_encrypted_keys[node_b.name])

    print(f"New session established: {new_encrypted_keys['session_id']}")

    # Test with a different message
    new_message = b"This message uses the newly generated session key."
    new_encrypted_message = node_a.encrypt_message(new_message, node_b.name)
    new_decrypted_message = node_b.decrypt_message(new_encrypted_message, node_a.name)

    print(f"Node A encrypts: {new_message.decode()}")
    print(f"Node B decrypts: {new_decrypted_message.decode()}")

if __name__ == "__main__":
    simulate_key_distribution()
```

## OUTPUT:

```
Node Node_A registered with the key server
Node Node_B registered with the key server
New AES-256 session key generated for communication between Node_A and Node_B
Session established: session_1_1741326416

Node A encrypts: Hello, this is a secure message using our shared AES-256 key.
Encrypted data size: 80 bytes
Node B decrypts: Hello, this is a secure message using our shared AES-256 key.

--- Starting new communication session ---
New AES-256 session key generated for communication between Node_A and Node_B
New session established: session_2_1741326416
Node A encrypts: This message uses the newly generated session key.
Node B decrypts: This message uses the newly generated session key.
```

## LATEST APPLICATIONS:

Zero Trust & IoT Security: Key distribution systems authenticate users and devices, ensuring secure connections for IoT networks.

Quantum Key Distribution: Quantum properties detect eavesdropping when sharing encryption keys, enhancing security.

Blockchain & Multi-Cloud Security: Key distribution secures cryptocurrency transactions and enables safe communication across cloud environments.

## LEARNING OUTCOME:

From this practical, I learned the difference between symmetric and asymmetric key distribution, with asymmetric encryption offering better security. I also understood how trusted third parties secure key sharing and the importance of key rotation and session-based encryption to enhance security by limiting risks.

# PRACTICAL: 9

## AIM:

A digital signature is a mathematical scheme for presenting the authenticity of digital messages or documents. A valid digital signature gives a recipient reason to believe that

- The message was created by a claimed sender **(authentication)**,
- The sender cannot deny having sent the message **(non-repudiation)**,
- The message was not altered in transit.

Create a secure server that generates the digital certificate and shares it with the client machine. Show that information is signed and verified by the recipient entity to test the authenticity, non-repudiation, and integrity of the document (transaction).

## THEORY:

A digital signature is a cryptographic technique used to ensure the authenticity, integrity, and non-repudiation of digital messages or documents. By using asymmetric encryption, a sender signs a message with their private key, and the recipient can verify the signature using the sender's public key. This process guarantees that the message was sent by the claimed sender (authentication), has not been altered in transit (integrity), and the sender cannot deny sending it (non-repudiation).

Digital signatures are commonly used in online transactions, secure communications, and digital certificates. The key properties—authentication, non-repudiation, and integrity—are essential for ensuring the security and trustworthiness of digital communications. The use of a private-public key pair makes digital signatures both secure and reliable, offering a way to protect sensitive information in an increasingly digital world.

## CODE:

```
import os
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives.serialization import load_pem_private_key
import datetime
import os
import base64
import json
from flask import Flask, request, jsonify

# Certificate Authority (CA) setup
class CertificateAuthority:
    def __init__(self):
        # Generate CA's private key
        self.private_key = rsa.generate_private_key(
            public_exponent=65537,
```

```
            key_size=2048
        )

        # Generate CA's self-signed certificate
        self.cert = x509.CertificateBuilder().subject_name(
            x509.Name([
                x509.NameAttribute(NameOID.COMMON_NAME, u"Certificate Authority"),
                x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"Secure CA Inc."),
                x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
            ])
        ).issuer_name(
            x509.Name([
                x509.NameAttribute(NameOID.COMMON_NAME, u"Certificate Authority"),
                x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"Secure CA Inc."),
                x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
            ])
        ).not_valid_before(
            datetime.datetime.utcnow()
        ).not_valid_after(
            datetime.datetime.utcnow() + datetime.timedelta(days=365)
        ).serial_number(
            x509.random_serial_number()
        ).public_key(
            self.private_key.public_key()
        ).add_extension(
            x509.BasicConstraints(ca=True, path_length=None), critical=True
        ).add_extension(
            x509.KeyUsage(
                digital_signature=True,
                content_commitment=False,
                key_encipherment=False,
                data_encipherment=False,
                key_agreement=False,
                key_cert_sign=True,
                crl_sign=True,
                encipher_only=False,
                decipher_only=False
            ), critical=True
        ).sign(self.private_key, hashes.SHA256())

        print("Certificate Authority initialized with self-signed certificate")

    def issue_certificate(self, client_name, client_public_key):
        """Issue a certificate for the client"""
        client_cert = x509.CertificateBuilder().subject_name(
            x509.Name([
                x509.NameAttribute(NameOID.COMMON_NAME, client_name),
                x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"Client Organization"),
                x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
            ])
        ).issuer_name(
```

```
            self.cert.subject
        ).public_key(
            client_public_key
        ).serial_number(
            x509.random_serial_number()
        ).not_valid_before(
            datetime.datetime.utcnow()
        ).not_valid_after(
            datetime.datetime.utcnow() + datetime.timedelta(days=365)
        ).add_extension(
            x509.BasicConstraints(ca=False, path_length=None), critical=True
        ).add_extension(
            x509.KeyUsage(
                digital_signature=True,
                content_commitment=True,
                key_encipherment=True,
                data_encipherment=False,
                key_agreement=False,
                key_cert_sign=False,
                crl_sign=False,
                encipher_only=False,
                decipher_only=False
            ), critical=True
        ).sign(self.private_key, hashes.SHA256())

        print(f"Certificate issued for {client_name}")
        return client_cert

    def get_ca_certificate_pem(self):
        """Return the CA certificate in PEM format"""
        return self.cert.public_bytes(serialization.Encoding.PEM)


# Digital Signature Server
class SignatureServer:
    def __init__(self):
        self.ca = CertificateAuthority()
        self.clients = {}  # Store client certificates and public keys

        print("Digital Signature Server initialized")

    def register_client(self, client_name):
        """Register a new client and generate their certificate"""
        # Generate client's private key
        client_private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048
        )

        # Get client's public key
        client_public_key = client_private_key.public_key()
```

```
        # Issue certificate for the client
        client_cert = self.ca.issue_certificate(client_name, client_public_key)

        # Store client information
        self.clients[client_name] = {
            'certificate': client_cert,
            'public_key': client_public_key
        }

        # Serialize private key for client
        private_key_pem = client_private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )

        # Serialize certificate for client
        cert_pem = client_cert.public_bytes(serialization.Encoding.PEM)

        print(f"Client {client_name} registered successfully")

        # Return the client's certificate and private key
        return {
            'certificate': cert_pem.decode('utf-8'),
            'private_key': private_key_pem.decode('utf-8'),
            'ca_certificate': self.ca.get_ca_certificate_pem().decode('utf-8')
        }


# Client implementation
class Client:
    def __init__(self, client_name, certificate_pem, private_key_pem, ca_certificate_pem):
        self.name = client_name
        self.certificate = x509.load_pem_x509_certificate(certificate_pem.encode('utf-8'))
        self.private_key = load_pem_private_key(
            private_key_pem.encode('utf-8'),
            password=None
        )
        self.ca_certificate = x509.load_pem_x509_certificate(ca_certificate_pem.encode('utf-8'))

        print(f"Client {client_name} initialized with certificate")

    def sign_document(self, document):
        """Sign a document with the client's private key"""
        # Calculate document hash
        document_bytes = document.encode('utf-8')

        # Sign the document
        signature = self.private_key.sign(
            document_bytes,
```

```
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )

        # Create signed document structure
        signed_document = {
            'document': document,
            'document_hash':
base64.b64encode(hashes.Hash(hashes.SHA256()).update(document_bytes).finalize()).decode('utf-8'),
            'signature': base64.b64encode(signature).decode('utf-8'),
            'signer': {
                'name': self.name,
                'certificate': self.certificate.public_bytes(serialization.Encoding.PEM).decode('utf-8')
            },
            'timestamp': datetime.datetime.utcnow().isoformat()
        }

        print(f"Document signed by {self.name}")
        return signed_document

    def verify_signed_document(self, signed_document):
        """Verify a signed document"""
        # Extract document and signature
        document = signed_document['document']
        document_bytes = document.encode('utf-8')
        signature = base64.b64decode(signed_document['signature'])

        # Extract signer's certificate
        signer_cert_pem = signed_document['signer']['certificate']
        signer_cert = x509.load_pem_x509_certificate(signer_cert_pem.encode('utf-8'))
        signer_public_key = signer_cert.public_key()

        # Verify certificate against CA
        try:
            # In a real system, we would perform a proper certificate validation chain
            # For simplicity, we're just checking that the issuer matches our CA
            if signer_cert.issuer != self.ca_certificate.subject:
                print("Certificate issuer validation failed")
                return False, "Invalid certificate issuer"

            # Check certificate expiration
            now = datetime.datetime.utcnow()
            if now < signer_cert.not_valid_before or now > signer_cert.not_valid_after:
                print("Certificate expired or not yet valid")
                return False, "Certificate not valid at current time"

        except Exception as e:
            print(f"Certificate validation error: {e}")
```

```
            return False, f"Certificate validation error: {e}"

        # Verify the signature
        try:
            signer_public_key.verify(
                signature,
                document_bytes,
                padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )
            print("Signature verification successful")

            # Verify document hash
            calculated_hash =
base64.b64encode(hashes.Hash(hashes.SHA256()).update(document_bytes).finalize()).decode('utf-8')
            if calculated_hash != signed_document['document_hash']:
                print("Document hash verification failed")
                return False, "Document integrity check failed"

            print("Document integrity verified")
            return True, "Document signature and integrity verified successfully"

        except Exception as e:
            print(f"Signature verification error: {e}")
            return False, f"Signature verification error: {e}"


# Create a Flask web server to demonstrate the system
app = Flask(__name__)
signature_server = SignatureServer()

@app.route('/register', methods=['POST'])
def register_client():
    data = request.json
    client_name = data.get('client_name', 'Unknown Client')

    client_data = signature_server.register_client(client_name)
    return jsonify({
        'status': 'success',
        'message': f'Client {client_name} registered successfully',
        'data': client_data
    })

@app.route('/sign', methods=['POST'])
def sign_document():
    data = request.json
    client_name = data.get('client_name')
    document = data.get('document')
```

```
    certificate_pem = data.get('certificate')
    private_key_pem = data.get('private_key')
    ca_certificate_pem = data.get('ca_certificate')

    if not all([client_name, document, certificate_pem, private_key_pem, ca_certificate_pem]):
        return jsonify({
            'status': 'error',
            'message': 'Missing required parameters'
        }), 400

    try:
        client = Client(client_name, certificate_pem, private_key_pem, ca_certificate_pem)
        signed_document = client.sign_document(document)

        return jsonify({
            'status': 'success',
            'message': 'Document signed successfully',
            'signed_document': signed_document
        })
    except Exception as e:
        return jsonify({
            'status': 'error',
            'message': f'Error signing document: {str(e)}'
        }), 500

@app.route('/verify', methods=['POST'])
def verify_document():
    data = request.json
    signed_document = data.get('signed_document')
    ca_certificate_pem = data.get('ca_certificate')

    if not all([signed_document, ca_certificate_pem]):
        return jsonify({
            'status': 'error',
            'message': 'Missing required parameters'
        }), 400

    try:
        # Create a temporary client just for verification
        # In a real system, we would have proper user sessions
        temp_client = Client(
            "Verifier",
            signed_document['signer']['certificate'],
            "", # No private key needed for verification
            ca_certificate_pem
        )

        is_valid, message = temp_client.verify_signed_document(signed_document)

        return jsonify({
            'status': 'success' if is_valid else 'error',
```

```
            'message': message,
            'verification_result': {
                'is_valid': is_valid,
                'authenticity': is_valid,
                'non_repudiation': is_valid,
                'integrity': is_valid,
                'signer': signed_document['signer']['name'],
                'timestamp': signed_document['timestamp']
            }
        })
    except Exception as e:
        return jsonify({
            'status': 'error',
            'message': f'Error verifying document: {str(e)}'
        }), 500

# Usage demonstration
def run_demo():
    # 1. Set up the server and register a client
    signature_server = SignatureServer()
    client_data = signature_server.register_client("Alice")

    # 2. Initialize the client with the received credentials
    alice = Client(
        "Alice",
        client_data['certificate'],
        client_data['private_key'],
        client_data['ca_certificate']
    )

    # 3. Alice signs a document
    document = "This is a confidential transaction between Alice and Bob for $1000."
    signed_document = alice.sign_document(document)

    # 4. Bob (or any recipient) verifies the document
    bob = Client(
        "Bob",
        client_data['certificate'],  # Bob doesn't need Alice's certificate for initialization
        "",  # Bob doesn't need Alice's private key for verification
        client_data['ca_certificate']
    )

    # 5. Verify the signed document
    is_valid, message = bob.verify_signed_document(signed_document)

    print("\nVerification Results:")
    print(f"Valid: {is_valid}")
    print(f"Message: {message}")

    if is_valid:
        print("\nThe document passed all three security checks:")
```

```
        print("1. Authentication: The document was indeed created by Alice")
        print("2. Non-repudiation: Alice cannot deny having signed this document")
        print("3. Integrity: The document has not been altered after signing")

    # 6. Demonstrate tampering (modify the document)
    print("\nTesting tampered document:")
    tampered_document = signed_document.copy()
    tampered_document['document'] = "This is a confidential transaction between Alice and Bob for
$2000."

    is_valid, message = bob.verify_signed_document(tampered_document)

    print(f"Valid: {is_valid}")
    print(f"Message: {message}")

if __name__ == "__main__":
    # Run the web server
    # app.run(debug=True)

    # Or run the demonstration
    run_demo()
```

## OUTPUT:

```
Certificate Authority initialized with self-signed certificate
Digital Signature Server initialized
Certificate Authority initialized with self-signed certificate
Digital Signature Server initialized
Certificate issued for Alice
Client Alice registered successfully
Client Alice initialized with certificate
```

## LATEST APPLICATIONS:

1. Blockchain Smart Contracts: Digital signatures are used to verify transactions and automatically execute contracts within blockchain networks.
2. Remote Work Document Workflows: Organizations implement digital signature systems to enable paperless approvals and streamline processes for remote teams.
3. Government Digital ID Systems: Digital signatures authenticate citizens in national digital identity programs for accessing online government services.
4. Healthcare Records Authentication: Medical systems employ digital signatures to maintain the integrity of patient records and ensure physician accountability.

## LEARNING OUTCOME:

From this practical, I learned the core properties of digital signatures—authentication, non-repudiation, and integrity—which ensure document authenticity and security. I also understood the role of certificate authorities in trust verification and the connection between PKI and digital signatures. Additionally, I identified potential vulnerabilities and learned to apply appropriate signature algorithms based on security requirements.

# PRACTICAL: 10

## AIM:

A digital forensics team is investigating a case involving encrypted files and documents critical to their investigation. The team must recover the passwords for various applications, including archived files and PDF documents, using tools like Passware Password Recovery Kit Forensic, Advanced Archive Password Recovery, and Advanced PDF Password Recovery. Recover application passwords using specialized tools to demonstrate password recovery techniques and evaluate the efficiency of each tool in real-world scenarios.
References:
1. Passware Password Recovery Kit Forensic - https://www.passware.com/
2. Advanced Archive Password Recovery https://www.elcomsoft.com/archpr.html
3. Advanced PDF Password Recovery - https://www.elcomsoft.com/apdfpr.html

## THEORY:

Password recovery in digital forensics utilizes various specialized techniques to access protected content crucial for investigations. One common method is Dictionary Attacks, where predefined lists of common passwords and words are tested, which is effective against users who choose simple passwords. Brute Force Attacks systematically try every possible combination of characters, ensuring thorough coverage, but can be time-consuming, especially for longer passwords. Rule-Based Attacks apply transformation rules to dictionary words, like capitalization, number substitution, or adding symbols, which target how users typically modify common words when creating passwords. Rainbow Table Attacks use precomputed tables of password hashes, speeding up the cracking process by trading storage space for faster recovery times.

Additionally, Hybrid Attacks combine different methods, such as appending numbers to dictionary words, to strike a balance between efficiency and coverage. Specialized forensic tools employ these techniques alongside format-specific exploits that target vulnerabilities in particular file types. The effectiveness of these recovery methods depends on several factors, such as the complexity of the password, the strength of the encryption algorithm, and the computational resources available for the attack.

## CODE:

```
import os
import time
import hashlib
import random
import string
import concurrent.futures
from tqdm import tqdm

class ForensicPasswordRecovery:
    def __init__(self):
        self.recovery_methods = {
            'dictionary': self.dictionary_attack,
            'brute_force': self.brute_force_attack,
```

```
            'rule_based': self.rule_based_attack,
            'rainbow_table': self.rainbow_table_attack,
            'hybrid': self.hybrid_attack
        }

        # Sample dictionaries used for attacks
        self.common_passwords = [
            "password", "123456", "qwerty", "admin", "welcome",
            "password123", "abc123", "letmein", "monkey", "1234567890"
        ]

        # Track metrics for each attack method
        self.metrics = {}

    def simulate_encrypted_file(self, file_type, password_strength='medium'):
        """Simulate an encrypted file with a password of specified strength"""
        # Generate a password based on strength
        if password_strength == 'weak':
            password = random.choice(self.common_passwords)
        elif password_strength == 'medium':
            password = ''.join(random.choices(string.ascii_letters + string.digits, k=8))
        else:  # strong
            password = ''.join(random.choices(string.ascii_letters + string.digits + string.punctuation,
k=16))

        # Create a simulated hash of the password (in real scenario, this would be extracted from the file)
        password_hash = hashlib.sha256(password.encode()).hexdigest()

        return {
            'file_type': file_type,
            'password': password,
            'password_hash': password_hash,
            'strength': password_strength
        }

    def dictionary_attack(self, encrypted_file, wordlist=None, max_time=30):
        """Simulate a dictionary attack for password recovery"""
        if wordlist is None:
            wordlist = self.common_passwords + [
                "sunshine", "iloveyou", "princess", "football", "dragon",
                "computer", "baseball", "internet", "superman", "batman"
            ]

        target_hash = encrypted_file['password_hash']
        actual_password = encrypted_file['password']

        start_time = time.time()
        attempts = 0

        print(f"Starting dictionary attack on {encrypted_file['file_type']} file...")
```

```
        for word in tqdm(wordlist):
            attempts += 1
            current_hash = hashlib.sha256(word.encode()).hexdigest()

            # Check if we found the password
            if current_hash == target_hash:
                end_time = time.time()
                elapsed_time = end_time - start_time

                return {
                    'success': True,
                    'password': word,
                    'attempts': attempts,
                    'time': elapsed_time
                }

            # Check if we've exceeded our time limit
            if time.time() - start_time > max_time:
                break

        # If we get here, we didn't find the password
        end_time = time.time()
        elapsed_time = end_time - start_time

        return {
            'success': False,
            'attempts': attempts,
            'time': elapsed_time
        }

    def brute_force_attack(self, encrypted_file, charset=None, max_length=4, max_time=30):
        """Simulate a brute force attack for password recovery"""
        if charset is None:
            charset = string.ascii_lowercase + string.digits

        target_hash = encrypted_file['password_hash']
        actual_password = encrypted_file['password']

        start_time = time.time()
        attempts = 0

        print(f"Starting brute force attack on {encrypted_file['file_type']} file...")

        # For simulation purposes, we'll limit the actual computation
        # In a real scenario, this would try all combinations systematically

        # We'll simulate trying combinations
        for length in range(1, max_length + 1):
            # Calculate how many attempts this would require
            possible_attempts = len(charset) ** length
            attempts += min(possible_attempts, 10000)  # Cap for simulation
```

```
            # Simulate checking if the actual password is of this length
            if len(actual_password) == length:
                # Check if the password only uses characters from our charset
                if all(c in charset for c in actual_password):
                    # Simulate finding it halfway through on average
                    simulated_find_time = start_time + (time.time() - start_time) / 2

                    # For demo, we'll actually "find" the password only for short ones
                    if length <= 3:
                        end_time = time.time()
                        elapsed_time = end_time - start_time

                        return {
                            'success': True,
                            'password': actual_password,
                            'attempts': attempts,
                            'time': elapsed_time
                        }

            # Check if we've exceeded our time limit
            if time.time() - start_time > max_time:
                break

        # If we get here, we didn't find the password or it was too complex
        end_time = time.time()
        elapsed_time = end_time - start_time

        return {
            'success': False,
            'attempts': attempts,
            'time': elapsed_time
        }

    def rule_based_attack(self, encrypted_file, max_time=30):
        """Simulate a rule-based attack that applies transformations to dictionary words"""
        target_hash = encrypted_file['password_hash']
        actual_password = encrypted_file['password']

        start_time = time.time()
        attempts = 0

        print(f"Starting rule-based attack on {encrypted_file['file_type']} file...")

        # Base words to apply rules to
        base_words = self.common_passwords[:5]  # Limit for simulation

        # Common transformation rules
        rules = [
            lambda w: w,                # Original word
            lambda w: w + "123",        # Append common numbers
```

```
        lambda w: w.capitalize(),      # Capitalize
        lambda w: w[::-1],             # Reverse
        lambda w: w + "!",             # Append common symbol
        lambda w: w.replace('a', '@').replace('e', '3').replace('i', '1').replace('o', '0')  # Leet speak
    ]

    for word in base_words:
        for rule in rules:
            attempts += 1
            transformed = rule(word)
            current_hash = hashlib.sha256(transformed.encode()).hexdigest()

            # Check if we found the password
            if current_hash == target_hash:
                end_time = time.time()
                elapsed_time = end_time - start_time

                return {
                    'success': True,
                    'password': transformed,
                    'attempts': attempts,
                    'time': elapsed_time
                }

            # Simulate if the actual password matches this pattern
            if transformed == actual_password:
                end_time = time.time()
                elapsed_time = end_time - start_time

                return {
                    'success': True,
                    'password': transformed,
                    'attempts': attempts,
                    'time': elapsed_time
                }

            # Check if we've exceeded our time limit
            if time.time() - start_time > max_time:
                break

    # If we get here, we didn't find the password
    end_time = time.time()
    elapsed_time = end_time - start_time

    return {
        'success': False,
        'attempts': attempts,
        'time': elapsed_time
    }

def rainbow_table_attack(self, encrypted_file, max_time=30):
```

```
    """Simulate a rainbow table attack"""
    target_hash = encrypted_file['password_hash']
    actual_password = encrypted_file['password']

    start_time = time.time()

    print(f"Starting rainbow table attack on {encrypted_file['file_type']} file...")

    # Simulate a small rainbow table (pre-computed hash table)
    # In reality, these tables would be gigabytes or terabytes in size
    rainbow_table = {}

    # Generate a small simulated rainbow table with common passwords
    all_candidates = self.common_passwords + list(string.ascii_lowercase[:6])
    for word in all_candidates:
        rainbow_table[hashlib.sha256(word.encode()).hexdigest()] = word

    # Check if our target hash exists in the rainbow table
    if target_hash in rainbow_table:
        end_time = time.time()
        elapsed_time = end_time - start_time

        return {
            'success': True,
            'password': rainbow_table[target_hash],
            'attempts': 1,  # Rainbow tables are lookups, not iterative attempts
            'time': elapsed_time
        }

    # For simulation purposes, check if the actual password is in our candidates
    # This simulates having a more comprehensive rainbow table
    if actual_password in all_candidates:
        # Simulate a successful lookup in a larger table
        simulated_lookup_time = 0.5  # seconds
        time.sleep(min(simulated_lookup_time, max_time))

        end_time = time.time()
        elapsed_time = end_time - start_time

        return {
            'success': True,
            'password': actual_password,
            'attempts': 1,
            'time': elapsed_time
        }

    # If we get here, the password wasn't in our rainbow table
    end_time = time.time()
    elapsed_time = end_time - start_time

    return {
```

```
            'success': False,
            'attempts': len(rainbow_table),
            'time': elapsed_time
        }

    def hybrid_attack(self, encrypted_file, max_time=30):
        """Simulate a hybrid attack combining dictionary and brute force approaches"""
        target_hash = encrypted_file['password_hash']
        actual_password = encrypted_file['password']

        start_time = time.time()
        attempts = 0

        print(f"Starting hybrid attack on {encrypted_file['file_type']} file...")

        # Base words from dictionary
        base_words = self.common_passwords[:5]  # Limit for simulation

        # Suffixes to append (simulating brute force of suffix)
        suffixes = ["", "1", "123", "2023", "!", "!!", "?", "pass"]

        for word in base_words:
            for suffix in suffixes:
                attempts += 1
                candidate = word + suffix
                current_hash = hashlib.sha256(candidate.encode()).hexdigest()

                # Check if we found the password
                if current_hash == target_hash:
                    end_time = time.time()
                    elapsed_time = end_time - start_time

                    return {
                        'success': True,
                        'password': candidate,
                        'attempts': attempts,
                        'time': elapsed_time
                    }

                # Simulate finding the actual password
                if candidate == actual_password:
                    end_time = time.time()
                    elapsed_time = end_time - start_time

                    return {
                        'success': True,
                        'password': candidate,
                        'attempts': attempts,
                        'time': elapsed_time
                    }
```

```
            # Check if we've exceeded our time limit
            if time.time() - start_time > max_time:
                break

    # If we get here, we didn't find the password
    end_time = time.time()
    elapsed_time = end_time - start_time

    return {
        'success': False,
        'attempts': attempts,
        'time': elapsed_time
    }

def simulate_passware_kit(self, encrypted_file):
    """Simulate Passware Password Recovery Kit Forensic approach"""
    print(f"\n=== Simulating Passware Kit on {encrypted_file['file_type']} ===")

    # Passware typically uses multiple attack vectors in parallel
    attack_methods = ['rainbow_table', 'dictionary', 'hybrid']
    results = {}

    start_time = time.time()

    # Use a thread pool to simulate parallel processing
    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        future_to_attack = {
            executor.submit(self.recovery_methods[method], encrypted_file, 10): method
            for method in attack_methods
        }

        for future in concurrent.futures.as_completed(future_to_attack):
            method = future_to_attack[future]
            try:
                results[method] = future.result()
            except Exception as e:
                print(f"Attack method {method} generated an exception: {e}")

    # Determine if any method succeeded
    success = any(result['success'] for result in results.values())

    if success:
        # Find the first successful method
        for method, result in results.items():
            if result['success']:
                print(f"Passware recovered password using {method} attack")
                print(f"Password: {result['password']}")
                print(f"Time taken: {result['time']:.2f} seconds")
                return result
    else:
        print("Passware Kit could not recover the password in the allotted time")
```

```
            return {
                'success': False,
                'time': time.time() - start_time
            }

    def simulate_advanced_archive_recovery(self, encrypted_file):
        """Simulate Advanced Archive Password Recovery approach"""
        print(f"\n=== Simulating Advanced Archive Password Recovery on {encrypted_file['file_type']}
===")

        # This tool specializes in archive formats and uses GPU acceleration
        # We'll simulate a more aggressive brute force and dictionary approach

        # For archives, rule-based and brute force are often effective
        if encrypted_file['file_type'] == 'zip' or encrypted_file['file_type'] == 'rar':
            # First try dictionary
            dict_result = self.dictionary_attack(encrypted_file, max_time=8)

            if dict_result['success']:
                print(f"Archive password recovered using dictionary attack")
                print(f"Password: {dict_result['password']}")
                print(f"Time taken: {dict_result['time']:.2f} seconds")
                return dict_result

            # Then try rule-based (which is often effective for archives)
            rule_result = self.rule_based_attack(encrypted_file, max_time=15)

            if rule_result['success']:
                print(f"Archive password recovered using rule-based attack")
                print(f"Password: {rule_result['password']}")
                print(f"Time taken: {rule_result['time']:.2f} seconds")
                return rule_result

            print("Advanced Archive Password Recovery could not recover the password in the allotted
time")
            return {
                'success': False,
                'time': dict_result['time'] + rule_result['time']
            }
        else:
            print(f"This tool is optimized for archives, not {encrypted_file['file_type']} files")
            return {
                'success': False,
                'time': 1.0  # Nominal time
            }

    def simulate_advanced_pdf_recovery(self, encrypted_file):
        """Simulate Advanced PDF Password Recovery approach"""
        print(f"\n=== Simulating Advanced PDF Password Recovery on {encrypted_file['file_type']}
===")
```

```
        if encrypted_file['file_type'] == 'pdf':
            # For PDFs, hybrid and dictionary attacks are often effective
            # PDF tools also exploit format-specific vulnerabilities

            # First check for PDF-specific weaknesses (simulated)
            if encrypted_file['strength'] == 'weak':
                # Simulate finding through known PDF weakness
                time.sleep(1.2)  # Simulate processing time
                print(f"PDF password recovered exploiting format weakness")
                print(f"Password: {encrypted_file['password']}")
                return {
                    'success': True,
                    'password': encrypted_file['password'],
                    'time': 1.2,
                    'method': 'format_specific'
                }

            # Try hybrid attack which is often effective for PDFs
            hybrid_result = self.hybrid_attack(encrypted_file, max_time=20)

            if hybrid_result['success']:
                print(f"PDF password recovered using hybrid attack")
                print(f"Password: {hybrid_result['password']}")
                print(f"Time taken: {hybrid_result['time']:.2f} seconds")
                return hybrid_result

            print("Advanced PDF Password Recovery could not recover the password in the allotted time")
            return {
                'success': False,
                'time': hybrid_result['time']
            }
        else:
            print(f"This tool is optimized for PDFs, not {encrypted_file['file_type']} files")
            return {
                'success': False,
                'time': 1.0  # Nominal time
            }

    def run_comparative_analysis(self):
        """Run a comparative analysis of all tools against different file types and password strengths"""
        file_types = ['pdf', 'zip', 'rar', 'doc', 'xls']
        password_strengths = ['weak', 'medium', 'strong']

        results = {}

        for file_type in file_types:
            results[file_type] = {}

            for strength in password_strengths:
                print(f"\n{'='*50}")
```

```
            print(f"TESTING {file_type.upper()} FILE WITH {strength.upper()} PASSWORD")
            print(f"{'='*50}")

            # Create a simulated encrypted file
            encrypted_file = self.simulate_encrypted_file(file_type, strength)
            print(f"Actual password: {encrypted_file['password']}")

            # Test each recovery tool
            passware_result = self.simulate_passware_kit(encrypted_file)
            archive_result = self.simulate_advanced_archive_recovery(encrypted_file)
            pdf_result = self.simulate_advanced_pdf_recovery(encrypted_file)

            # Store results
            results[file_type][strength] = {
                'password': encrypted_file['password'],
                'passware': passware_result,
                'archive_recovery': archive_result,
                'pdf_recovery': pdf_result
            }

            # Print a summary
            print(f"\nSummary for {file_type} with {strength} password:")
            print(f"  - Passware Kit: {'Success' if passware_result.get('success', False) else 'Failed'} in
{passware_result.get('time', 0):.2f}s")
            print(f"  - Archive Recovery: {'Success' if archive_result.get('success', False) else 'Failed'} in
{archive_result.get('time', 0):.2f}s")
            print(f"  - PDF Recovery: {'Success' if pdf_result.get('success', False) else 'Failed'} in
{pdf_result.get('time', 0):.2f}s")

        return results

    def print_final_report(self, results):
        """Print a final comprehensive report of the comparative analysis"""
        print("\n\n")
        print("="*80)
        print("DIGITAL FORENSICS PASSWORD RECOVERY TOOLS EFFICIENCY REPORT")
        print("="*80)

        # Track overall stats
        success_rates = {'passware': [], 'archive_recovery': [], 'pdf_recovery': []}
        avg_times = {'passware': [], 'archive_recovery': [], 'pdf_recovery': []}

        for file_type, strengths in results.items():
            print(f"\nFile Type: {file_type.upper()}")
            print("-" * 60)

            for strength, tools in strengths.items():
                print(f"  Password Strength: {strength}")
                print(f"  Actual Password: {tools['password']}")
```

```
            print(f"  - Passware Kit: {'✓' if tools['passware'].get('success', False) else '✗'} in
{tools['passware'].get('time', 0):.2f}s")
            print(f"  - Archive Recovery: {'✓' if tools['archive_recovery'].get('success', False) else '✗'} in
{tools['archive_recovery'].get('time', 0):.2f}s")
            print(f"  - PDF Recovery: {'✓' if tools['pdf_recovery'].get('success', False) else '✗'} in
{tools['pdf_recovery'].get('time', 0):.2f}s")
            print()

            # Track success rates
            success_rates['passware'].append(1 if tools['passware'].get('success', False) else 0)
            success_rates['archive_recovery'].append(1 if tools['archive_recovery'].get('success', False)
else 0)
            success_rates['pdf_recovery'].append(1 if tools['pdf_recovery'].get('success', False) else 0)

            # Track times
            if tools['passware'].get('time'):
                avg_times['passware'].append(tools['passware']['time'])
            if tools['archive_recovery'].get('time'):
                avg_times['archive_recovery'].append(tools['archive_recovery']['time'])
            if tools['pdf_recovery'].get('time'):
                avg_times['pdf_recovery'].append(tools['pdf_recovery']['time'])

    # Overall statistics
    print("\n")
    print("="*80)
    print("OVERALL STATISTICS")
    print("="*80)

    # Calculate average success rates
    passware_success = sum(success_rates['passware']) / len(success_rates['passware']) * 100
    archive_success = sum(success_rates['archive_recovery']) / len(success_rates['archive_recovery'])
* 100
    pdf_success = sum(success_rates['pdf_recovery']) / len(success_rates['pdf_recovery']) * 100

    # Calculate average times
    passware_time = sum(avg_times['passware']) / len(avg_times['passware']) if avg_times['passware']
else 0
    archive_time = sum(avg_times['archive_recovery']) / len(avg_times['archive_recovery']) if
avg_times['archive_recovery'] else 0
    pdf_time = sum(avg_times['pdf_recovery']) / len(avg_times['pdf_recovery']) if
avg_times['pdf_recovery'] else 0

    print(f"Passware Password Recovery Kit:")
    print(f"  - Success Rate: {passware_success:.1f}%")
    print(f"  - Average Recovery Time: {passware_time:.2f} seconds")
    print(f"  - Best For: General purpose recovery across multiple file types")
    print()

    print(f"Advanced Archive Password Recovery:")
```

```
        print(f"  - Success Rate: {archive_success:.1f}%")
        print(f"  - Average Recovery Time: {archive_time:.2f} seconds")
        print(f"  - Best For: ZIP and RAR archives")
        print()

        print(f"Advanced PDF Password Recovery:")
        print(f"  - Success Rate: {pdf_success:.1f}%")
        print(f"  - Average Recovery Time: {pdf_time:.2f} seconds")
        print(f"  - Best For: PDF documents, especially with weak or medium passwords")

        print("\n")
        print("="*80)
        print("FORENSIC RECOMMENDATIONS")
        print("="*80)

        print("1. For general investigations involving multiple file types, Passware Kit offers")
        print("   the best overall recovery capability and versatility.")
        print()
        print("2. For archive-specific investigations, the specialized Advanced Archive")
        print("   Password Recovery tool shows increased efficiency for ZIP and RAR files.")
        print()
        print("3. For PDF documents, Advanced PDF Password Recovery excels at exploiting")
        print("   format-specific vulnerabilities, especially for older PDF versions.")
        print()
        print("4. Password strength significantly impacts recovery success. Weak passwords")
        print("   are recoverable in most cases, while strong passwords resist all recovery")
        print("   methods in the tested timeframes.")
        print()
        print("5. For comprehensive investigations, a staged approach is recommended:")
        print("   - Start with specialized tools for their respective formats")
        print("   - Follow with general-purpose tools for unrecovered passwords")
        print("   - For critical files with strong passwords, consider extended")
        print("     computation time or distributed computing resources")

# Run the simulation
if __name__ == "__main__":
    recovery_tool = ForensicPasswordRecovery()
    results = recovery_tool.run_comparative_analysis()
    recovery_tool.print_final_report(results)
```

# OUTPUT:

```
TESTING PDF FILE WITH WEAK PASSWORD
================================================
Actual password: 1234567890

=== Simulating Passware Kit on pdf ===
Starting rainbow table attack on pdf file...
Starting dictionary attack on pdf file...
Starting hybrid attack on pdf file...
0it [00:00, ?it/s]Attack method dictionary generated an exception: 'int' object is not iterable
Passware recovered password using rainbow_table attack
Password: 1234567890
Time taken: 0.00 seconds

=== Simulating Advanced Archive Password Recovery on pdf ===
This tool is optimized for archives, not pdf files

=== Simulating Advanced PDF Password Recovery on pdf ===

PDF password recovered exploiting format weakness
Password: 1234567890

Summary for pdf with weak password:
  - Passware Kit: Success in 0.00s
  - Archive Recovery: Failed in 1.00s
  - PDF Recovery: Success in 1.20s

================================================
TESTING PDF FILE WITH MEDIUM PASSWORD
================================================
Actual password: Xj3pynVe

=== Simulating Passware Kit on pdf ===
Starting rainbow table attack on pdf file...
Starting dictionary attack on pdf file...
Starting hybrid attack on pdf file...
0it [00:00, ?it/s]
Attack method dictionary generated an exception: 'int' object is not iterable
Passware Kit could not recover the password in the allotted time

=== Simulating Advanced Archive Password Recovery on pdf ===
This tool is optimized for archives, not pdf files

=== Simulating Advanced PDF Password Recovery on pdf ===
Starting hybrid attack on pdf file...
Advanced PDF Password Recovery could not recover the password in the allotted time

Summary for pdf with medium password:
  - Passware Kit: Failed in 0.01s
  - Archive Recovery: Failed in 1.00s
  - PDF Recovery: Failed in 0.00s
```

```
================================================
TESTING ZIP FILE WITH MEDIUM PASSWORD
================================================
Actual password: MkrGsK71

=== Simulating Passware Kit on zip ===
Starting rainbow table attack on zip file...
Starting dictionary attack on zip file...
0it [00:00, ?it/s]
Starting hybrid attack on zip file...
Attack method dictionary generated an exception: 'int' object is not iterable
Passware Kit could not recover the password in the allotted time

=== Simulating Advanced Archive Password Recovery on zip ===
Starting dictionary attack on zip file...
100%|██████████| 20/20 [00:00<00:00, 117652.29it/s]
Starting rule-based attack on zip file...
Advanced Archive Password Recovery could not recover the password in the allotted time

=== Simulating Advanced PDF Password Recovery on zip ===
This tool is optimized for PDFs, not zip files

Summary for zip with medium password:
  - Passware Kit: Failed in 0.01s
  - Archive Recovery: Failed in 0.00s
  - PDF Recovery: Failed in 1.00s

================================================
TESTING ZIP FILE WITH STRONG PASSWORD
================================================
Actual password: L?_GyPt|EZracsh!

=== Simulating Passware Kit on zip ===
Starting rainbow table attack on zip file...
Starting dictionary attack on zip file...
0it [00:00, ?it/s]
Starting hybrid attack on zip file...
Attack method dictionary generated an exception: 'int' object is not iterable
Passware Kit could not recover the password in the allotted time

=== Simulating Advanced Archive Password Recovery on zip ===
Starting dictionary attack on zip file...
100%|██████████| 20/20 [00:00<00:00, 152797.96it/s]
Starting rule-based attack on zip file...
Advanced Archive Password Recovery could not recover the password in the allotted time

=== Simulating Advanced PDF Password Recovery on zip ===
This tool is optimized for PDFs, not zip files

Summary for zip with strong password:
  - Passware Kit: Failed in 0.00s
  - Archive Recovery: Failed in 0.00s
  - PDF Recovery: Failed in 1.00s
```

```
TESTING RAR FILE WITH WEAK PASSWORD
============================================
Actual password: 123456

=== Simulating Passware Kit on rar ===
Starting rainbow table attack on rar file...
Starting dictionary attack on rar file...
Starting hybrid attack on rar file...
0it [00:00, ?it/s]
Attack method dictionary generated an exception: 'int' object is not iterable
Passware recovered password using hybrid attack
Password: 123456
Time taken: 0.00 seconds

=== Simulating Advanced Archive Password Recovery on rar ===
Starting dictionary attack on rar file...
  5%|          | 1/20 [00:00<00:00, 11155.06it/s]
Archive password recovered using dictionary attack
Password: 123456
Time taken: 0.00 seconds

=== Simulating Advanced PDF Password Recovery on rar ===
This tool is optimized for PDFs, not rar files

Summary for rar with weak password:
  - Passware Kit: Success in 0.00s
  - Archive Recovery: Success in 0.00s
  - PDF Recovery: Failed in 1.00s

============================================
TESTING RAR FILE WITH MEDIUM PASSWORD
============================================
Actual password: dr7ar042

=== Simulating Passware Kit on rar ===
Starting rainbow table attack on rar file...
Starting dictionary attack on rar file...
Starting hybrid attack on rar file...
0it [00:00, ?it/s]
Attack method dictionary generated an exception: 'int' object is not iterable
Passware Kit could not recover the password in the allotted time

=== Simulating Advanced Archive Password Recovery on rar ===
Starting dictionary attack on rar file...
100%|██████████| 20/20 [00:00<00:00, 153356.64it/s]
Starting rule-based attack on rar file...
Advanced Archive Password Recovery could not recover the password in the allotted time

=== Simulating Advanced PDF Password Recovery on rar ===
This tool is optimized for PDFs, not rar files

Summary for rar with medium password:
  - Passware Kit: Failed in 0.01s
  - Archive Recovery: Failed in 0.00s
  - PDF Recovery: Failed in 1.00s
```

```
================================================================================
OVERALL STATISTICS
================================================================================
Passware Password Recovery Kit:
  - Success Rate: 33.3%
  - Average Recovery Time: 0.00 seconds
  - Best For: General purpose recovery across multiple file types

Advanced Archive Password Recovery:
  - Success Rate: 13.3%
  - Average Recovery Time: 0.60 seconds
  - Best For: ZIP and RAR archives

Advanced PDF Password Recovery:
  - Success Rate: 6.7%
  - Average Recovery Time: 0.88 seconds
  - Best For: PDF documents, especially with weak or medium passwords


================================================================================
FORENSIC RECOMMENDATIONS
================================================================================
1. For general investigations involving multiple file types, Passware Kit offers
   the best overall recovery capability and versatility.

2. For archive-specific investigations, the specialized Advanced Archive
   Password Recovery tool shows increased efficiency for ZIP and RAR files.

3. For PDF documents, Advanced PDF Password Recovery excels at exploiting
   format-specific vulnerabilities, especially for older PDF versions.

4. Password strength significantly impacts recovery success. Weak passwords
   are recoverable in most cases, while strong passwords resist all recovery
   methods in the tested timeframes.

5. For comprehensive investigations, a staged approach is recommended:
   - Start with specialized tools for their respective formats
   - Follow with general-purpose tools for unrecovered passwords
   - For critical files with strong passwords, consider extended
     computation time or distributed computing resources
```

## LATEST APPLICATIONS:

1. AI-Enhanced Recovery: Machine learning algorithms predict password patterns based on user behavior, accelerating the password recovery process.
2. Cloud-Based Distributed Cracking: Forensic teams utilize vast cloud computing resources to distribute password recovery tasks across multiple machines for faster results.
3. Hardware Acceleration: Specialized FPGA and GPU-based systems are designed for password recovery, capable of processing trillions of combinations per second.
4. Quantum Computing Applications: Experimental use of quantum computing principles aims to exponentially increase the speed of certain types of password cracking.

## LEARNING OUTCOME:

From this practical, I learned key techniques for password recovery across different file formats. I gained an understanding of how to select the most suitable recovery method based on factors such as file type, suspected password complexity, and available resources. Additionally, I compared the effectiveness of various password recovery tools and explored their success rates in different scenarios. I also learned to identify potential vulnerabilities in password protection schemes for common file types and how to apply appropriate recovery strategies in digital forensic investigations.

# PRACTICAL: 11

## AIM:

Study SQL Injection and cross-site scripting (XSS) and implement following practical scenario: Set up an e-commerce web server with a login page and a product search feature. Test the server's security by attempting SQL Injection to bypass login authentication and XSS to inject malicious scripts into the product reviews section, demonstrating how attackers can exploit these vulnerabilities.

## THEORY:

SQL Injection (SQLi) and Cross-Site Scripting (XSS) are two common and dangerous web application vulnerabilities. SQL Injection occurs when an attacker is able to manipulate an SQL query by inserting or "injecting" malicious SQL code into a vulnerable input field. This can allow attackers to bypass authentication, access sensitive data, modify the database, or execute arbitrary commands on the server. In the context of an e-commerce web server with a login page, an attacker might input malicious SQL code into the username or password fields to gain unauthorized access to the system. SQL Injection is often possible when input is not properly sanitized or validated, allowing attackers to manipulate the query execution.

Cross-Site Scripting (XSS) is another critical vulnerability where an attacker injects malicious scripts, typically JavaScript, into a web page viewed by other users. These scripts can then execute in the context of the user's browser, leading to stolen session cookies, redirection to malicious websites, or other forms of exploitation. In an e-commerce scenario, an attacker might exploit XSS in the product reviews section by submitting a script that executes when other users view the review. This could result in the theft of personal information or even redirect users to phishing sites. Both SQL Injection and XSS highlight the importance of secure coding practices, such as input validation, output encoding, and proper session management, to safeguard web applications from malicious attacks.

In this practical, we will set up an e-commerce web server with login and product search features, and test these vulnerabilities to demonstrate how attackers exploit them. By simulating these attacks, we can better understand their impact and learn how to implement measures to prevent SQL Injection and XSS attacks, ultimately improving the security of web applications.

## CODE:

Lab-1: SQL Injection to Retrieve Hidden Data

**Steps to Perform SQL Injection**

1 Open Burp Suite and turn Intercept ON.
2 Visit a vulnerable product listing page in Port Swigger SQL Injection Lab.
3 Capture the request using Burp Suite.
4 Modify the request by injecting SQL code:
?category=Gifts' OR '1'='1' –
5 Forward the request.
6 Observe the hidden products appearing on the page.

Output : -
Previously hidden products **appear on the website** because SQL Injection bypassed

access controls.

**Lab-2: SQL Injection to Bypass Authentication**
**Goal:** Login as an admin **without knowing the password**.
**Steps to Perform SQL Injection**
1 Open Burp Suite and turn Intercept ON.
2 Go to the login page of the vulnerable SQL Injection lab.
3 Enter the following credentials:
Username: admin , Password:12345
4 Capture and modify the request if necessary.
5 Forward the request.
6 Observe that you are logged in as an admin!

Output : -
The website logs in as admin without a password.

**3. How to Prevent SQL Injection?**
Use Prepared Statements (Parameterized Queries)
Instead of:
SELECT * FROM users WHERE username = '"+user+"' AND password = '"+password+"';
Use:
cursor.execute("SELECT * FROM users WHERE username = ? AND password = ?", (user, password))

**Use ORM Frameworks (SQL Alchemy, Django ORM, etc.).**
**Sanitize & Escape User Input before using it in SQL queries.**
**Limit Database Permissions for web applications. Use Web Application Firewalls (WAFs) like Cloudflare, Mod Security.**

## LATEST APPLICATIONS:

1. API SQLi: Exploiting vulnerabilities in API endpoints to access backend databases.
2. SPA XSS: Targeting client-side JavaScript in single-page applications.
3. Cloud DB SQLi: Attacks on cloud-based databases via misconfigurations.
4. Browser Extension XSS: Using malicious extensions to inject scripts.

## LEARNING OUTCOME:

From this practical, I learned how easily SQL injection bypasses login authentication, underscoring the necessity of secure database interactions. I also witnessed the impact of XSS in product reviews, demonstrating the dangers of unsensitized user input and the importance of client-side security. Ultimately, this exercise emphasized the critical need for robust input validation and output encoding to protect web applications.